University of Oslo
Department of Informatics

Issues Concerning Parameter Sets in Apotram

Anne Gro Kjølstad

Cand Scient Thesis

May 7 2001

Preface

This thesis is written at the University of Oslo—Department of Informatics—as a part of the cand. scient. degree. The degree is worth five years of full time studies, where approximately one year is devoted to this thesis.

My interest in database systems led me to contact Ole Jørgen Anfindsen who is an expert in transaction theory. His special research area is the transaction model Apotram, supporting cooperation among long lived transactions. It is a new research area with several open issues. Fortunately, Anfindsen was willing to supervise me through some of these issues. The results of my work is presented in this thesis.

I want to thank my supervisor for his pedagogical supervising, constructive criticism, for giving me a summer job in Apotram AS, and for nominating me as a student participant at the year 2000 conference on Persistent Object Systems (POS9).

I also want to thank my family, my fiancé's family, and all my friends for their support.

Finally, I want to thank my fiancé Pål Sommerhein for his love, support, and for setting up a comfortable and efficient technical working environment both at home and at the university.

Oslo, May 7 2001

Anne Gro Kjølstad

Abstract

One of the most important features of database systems is *transaction management*. Transaction management ensures *concurrency-* and *recovery* control.

In conventional database systems, *classical* transaction management ensures the *ACID* properties of transactions which are of utmost importance to many applications like banking, reservation systems, and bookkeeping.

However, there are *advanced* applications where classical transaction management is too restrictive. This is because classical transaction management enforces *total isolation* among transactions, while to some advanced applications it is important to let transactions *cooperate* by *sharing data* and do *teamwork*. Examples of applications for which the classical transaction management is too restricitive are CAD/CAM, publishing, and document handling.

Apotram is an *advanced transaction model* developed in order to meet the cooperation requirements of advanced applications. It was first presented in the PhD thesis of Ole Jørgen Anfindsen (Anfindsen, 1997). *Nested databases* in Apotram enable transactions to do teamwork, and *parameter sets* are used to customize the notion of conflict and to communicate information about uncommitted shared data to transactions.

In this thesis I have written an introduction to database systems and classical transaction management, a motivation for why classical transaction management sometimes is too restrictive, and a presentation of the advanced transaction model Apotram. This is the necessary background for the parameter set issues that are presented and discussed in chapter 5 and chapter 6.

Contents

Chapter 1

Databases

In this chapter it is given a brief overview of what database systems are, and a motivation for why database systems are attractive.

1.1 What is a Database System?

The following about databases and database managament systems is based on (Elmasri and Navathe, 1994, pages 1–2) and (Normann and Ressem, 1998, page 6).

1.1.1 Database

A database is a collection of related data. The related data originate from a *miniworld* or a *Universe of Discourse (UoD)*. Examples of UoDs are airline companies, libraries, dentist practices, railway companies, warehouse companies, hospitals, organizations, families, and schools.

We will next consider an airline company. All the information that the airline considers important can be stored in the database. E.g. information about employees, salaries, departments, flights, customers, and relations among these.

If there are changes in the miniworld or UoD, then changes must be reflected in the actual database. A change in the airline company's UoD can be that a new customer wants to make a flight reservation, and in order to reflect the change in the database, a reservation clerk must register information about the customer and the reservation in the database.

The collection of data in a database is logically coherent with some inherent meaning, and there is a purpose of the database. It has an intended group of users, e.g. all employees in an airline company. These users have some preconceived applications in which they are interested. Examples of such applications are to make reservations on flights, to change the address of an employee, to insert a new department, to delete an employee, and to search for information about flights.

1.1.2 Database Management System (DBMS)

A DBMS is a collection of programs that enable users to create and maintain a database. In addition to let transactions read and write data in a certain database, some of the DBMS's most important features are listed below.

- It must handle conflicts that occur when many application programs use the database at the same time. This is known as *concurrency control*.
- It must do recovery in case of failures. This means that a DBMS must ensure that every *logical unit* of read and write operations must be executed as a unit. Then *all* operations should be executed or *none*. This is called *recovery control*.
- It must ensure that users have permission to access what they try to access. This is known as *authorization*.
- It runs processes for back up, load/unload, and provides interfaces for programs written in many different programming languages like C, Smalltalk, Java, and Fortran.

This thesis is concerned with *transaction management*, which has the responsibility of the *concurrency-* and *recovery control* parts of a DBMS.

The database, and the database management system make together up a *database system*.

1.2 Why are Database Systems Attractive?

Database systems vary and their advantages are absolutely dependent of what features their DBMSs offer. But below I present some of the advantages that database systems *can* have, compared to conventional file-processing systems. The presentation is based on the chapter *Introduction* in (Silberschatz et al., 1997) and the chapter *Databases and Database Users* in (Elmasri and Navathe, 1994).

Please note that I only consider centralized database systems (CDBS) in this thesis.

- *A DBS reduces data redundancy, and inconsistency caused by data redundancy.* In a centralized database system the data is stored once and the storage capasity needed is reduced. When data is updated this is done only *once* in a CDBS. This means that it takes less effort to do the updating, and that the danger of having inconsistency among several copies of the same data is eliminated.

  In some cases, *controlled redundancy* may be useful. Then *integrity constraints* can be defined, and the DBMS will automatically enforce

these. For example, if a student's name is stored twice in a database, then an integrity constraint can state that these two names must be identical.

- *A DBMS has meta-data.* A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database, called *meta-data*. The meta-data is contained in the *system catalog*. This catalog is used by the DBMS software and occasionally by database users, who need information about the database structure. Whereas file-processing software can only access specific files, DBMS software can access diverse databases by extracting the database definitions from the catalog, and then use these definitions.

- *A DBS offers data abstraction.* In traditional file processing, the structure of data files is embedded in the access programs. Then any changes to the structure of a file may require changes in all programs that access this file. By contrast, *DBMS access programs* are written *independently* of any specific files because the structure of data files is stored in the DBMS catalog separately from the access programs. This means that the DBMS provides users with a *conceptual* representation of data, which does not include many of the details about *how* the data is stored.

  Informally, a *data model* is a type of data abstraction that is used to provide this conceptual representation. Examples of data models are the *relational model*, the *hierarchical model*, the *network model*, and the *object-oriented model*.

- *A DBMS makes it easy to access and manipulate data.* In addition to access data through application programs, a user can specify what he or she wants to access in a so called *query language*, or by navigating in the data structure. This is important because it enables data to be accessed in many different ways without first having a proper application program to use.

- *A DBMS should handle integrity constraints.* Most database applications have certain *integrity constraints* that must hold for the data. A DBMS should provide capabilities for defining and enforcing such constraints.

- *Providing backup and recovery.* A DBMS must provide facilities for *recovering* from hardware or software failures. The *backup and recovery subsystem* of the DBMS is responsible for recovery. If the computer system fails in the middle of a complex update program, then the recovery subsystem is responsible for restoring the database to the

state it was in before the program started to execute. Alternatively, the recovery subsystem can ensure that the program is resumed from the point where it was interrupted in order to record its full effect in the database.

- *The DBMS offers concurrency control.* A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data used by multiple applications is to be integrated and maintained in *one single* database, due to performance reasons. *Concurrency control* must be offered in order to ensure that the simultaneously accesses do not result in inconsistent databases.

- *Security control.* In a DBS, security control is offered. When multiple users share a database, it is likely that some users will not be authorized to access all data stored in the database. Therefore, it is important that the type of access operation, retrieval or update, is controlled by the DBMS before the operation is actually performed.

Not all applications will be better off with a database system instead of a conventional file system. This is due to the extra overhead a DBMS introduces. More about this can be found in (Elmasri and Navathe, 1994, pages 16–17).

1.3 Summary

In this chapter the concepts database, database management system, and database system have been introduced.

In addition some of the advantages of database systems have been outlined. Among these advantages are *concurrency- and recovery control.* Concurrency- and recovery control will be considered in more detail in chapter 2.

Chapter 2

Classical Transactions

In this chapter it is motivated why classical transaction theory was developed, and it is given an overview of what classical transaction theory is.

2.1 Transactions

2.1.1 Why was Transaction Theory Developed?

In a lot of classical database applications, it is important that users always have a correct view of the actual database. That is, the database must be *consistent* with a miniworld or UoD as discussed in chapter 1.

In the following example, a classical database application is described, namely a flight reservation application. The example illustrates an anomaly known as *the lost update*. The anomaly can happen if concurrent executions of applications are not controlled by the DBM in a proper way.

The Lost Update

Consider an airline company that has an online reservation system. The database contains important information, e.g. about *how many available seats* there are on each flight.

Imagine that two reservation clerks, Bob and Alice, will at the *same* time invoke an application that accesses the airline's database. They both want to make reservations at the *same* flight.

We will only consider how the application updates the count of available seats, but note that the application could have registered data about passengers and more as well. We assume that the seat numbers will be delivered when the actual passengers check in.

Bob's application reads how many seats that are available on the actual flight and gets the answer two. Right afterwards Alice's application reads too, and it gets the same answer. Then Bob makes a reservation for two people and writes back to the database that there are zero seats available.

Alice makes a reservation for one person and writes back to the database that there is one seat available. Then Bob's reservation is *lost*, and consequently the database has an *inconsistent* state. This is because there is registered in the database that one seat is available, while the flight is *actually* overbooked by one person in the UoD.

The consequence of this inconsistency would be that one person came to the airport, checking in for the flight in question, just to find out that there were no seats available for him or her. The person would probably not like the situation and neither would the airline company.

In this reservation application it is obviously very important that users have a *correct view* of the database at any time.

Error due to Concurrency

In *the lost update* example it is assumed that the application was coded correctly. The source of the inconsistency or error is *concurrent execution* of two clerks' programs in an *uncontrolled* manner.

Concurrent execution means that operations belonging to different programs are *interleaved*. Then instead of first executing *all* operations belonging to one program, then executing *all* operations belonging to the next program, and so on, the operations are mixed in order to obtain better performance.

When it is desirable to execute programs concurrently, it is important that the system offers *concurrency control*.

The lost update is only one example of what can go wrong when a database system allows concurrent execution *without* concurrency control. Other well known anomalies are *dirty read*, *the inconsistent analysis problem*, and *unrepeatable read*. These anomalies are discussed in e.g. (Bernstein et al., 1987, pages 11–13, 64-66), (Gray and Reuter, 1993, pages 380–381), and (Anfindsen, 1997, pages 34–35).

Avoiding that programs interfer with each other is called *the concurrency control problem* (Bernstein et al., 1987, page iii).

Error due to Failure

(Bernstein et al., 1987, page iii–iv) write:

> *Computer systems are subject to many types of failures. Operating systems fail, as does the hardware on which they run. When a failure occurs, one or more application programs may be interrupted in midstream. Since the program was written to be correct only under the assumption that it executed in its entirety, an interrupted execution can lead to incorrect results. For example, a money transfer application may be interrupted by a failure after*

debiting one account but before crediting the other. Avoiding such incorrect results due to failures is called the recovery problem.

An Important Assumption

If it is assumed that applications that access a database are tested and proved to be correct, then the errors that can occur will either be due to *concurrency* or to *failures* (Bernstein et al., 1987, page iii).

What is Transaction Theory?

Transaction theory is concerned with *solving* the concurrency control problem and the recovery problem, in order to keep databases correct. This is important, as we have seen above, because database systems must be able to give users a correct view of it.

We will next consider what a transaction actually is. Later in section 2.2 and section 2.3, we will se how the concurrency control problem and the recovery problem can be solved.

2.1.2 The ACID Properties

If we consider *the lost update* scenario where Bob and Alice made reservations, then we see that the application they used represents a *logical unit of work (LUW)*. The application first reads the count of seats available and then updates it.

There are certain properties LUWs should have in order to be *transactions*. These properties are *Atomicity*, *Consistency*, *Isolation*, and *Durability (ACID)*. The following description of the ACID properties is inspired by (Anfindsen, 1997, page 5).

- *Atomicity. All* operations of a LUW must be executed or *none at all*, and the user must, whatever happens, know which state he or she is in.

- *Consistency.* A LUW reaching its normal end (end of transaction), thereby comitting its results, *preserves the consistency* of the database. In other words, each successful LUW *by definition* commits only legal results. This means that it is *assumed* that programs which access the database have correct logic. This condition is necessary for the fourth property, durability.

  If the database is initially consistent, if no concurrency is allowed, and if no failure occurs, then a transaction takes the database to a new consistent state.

The DBMS cannot fully control whether the logic in programs are correct or not even though it can enforce some integrity constraints. Therefore this responsibility remains with application programmers.

- *Isolation.* Events within a LUW must be *hidden* from other LUW's running concurrently. Anomalies that can occur if LUWs are not hidden from each other are *the lost update*, *dirty read*, *the inconsistent analysis problem*, and *unrepetable read*. The techniques that achieve isolation are known as *concurrency control* (or synchronization).

- *Durability.* Once a LUW has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions. Therefore the user must get a guarantee that the things the system says have happenend actually have happened. Since, by definition, each LUW is correct, the effects of an inevitable incorrect LUW (i.e., the LUW has a logical error) can only be removed by counterLUWs.

Definition of a Classical Transaction

When a LUW has the ACID properties, it is considered a *transaction*.

A transaction is built of the database operations *begin*, *read*, *write*, *commit*, and *abort*. It always starts with *begin* and always ends with *abort* or *commit*.

The Operations *begin*, *read*, *write*, *commit*, and *abort*

- *begin.* This operation is used to begin a transaction. If a transaction t_i does this operation, then it is denoted t_i(b).

- *read.* This operation is used by a transaction to read a data item. If a transaction t_i reads a data item x, then it is denoted t_i(r, x).

- *write.* This operation is used by a transaction to write (or update) a data item. If a transaction t_i writes the value v into a data item x, then it is denoted t_i(w, x, v), or simply t_i(w, x) if the value written is ignored.

- *commit.* This operation is used by a transaction when it wants to commit its results to the database. When a transaction gets a message from the system that its commit is accepted, then it can trust that its effects in the database are durable. If a transaction t_i does commit, then this is denoted t_i(c).

- *abort.* This operation is used by a transaction when it wants to cancel its work, or it can be done by the system in case of e.g. failures. When a transaction gets a message from the system that it is aborted, then

it can trust that all its effects in the database are wiped out. To wipe out a transaction's effects is called a *roll back*. If a transaction t_i is aborted, then this is denoted t_i(a).

What a transaction reads or writes are *data items* or *data objects*. E.g. the *count of available seats* on a flight is an item or an object. I do not specify the size of a data item or a data object here because it is not essential for the discussion.

Next we will consider how the concurrency control problem can be solved.

2.2 Concurrency Control

The goal of concurrency control is to enable systems to *control the interleaved execution* of several programs that run concurrently. This is, as we saw above, in order to always give users a *correct view* of the database.

An idea that leads to a way of solving the concurrency control problem is that every program must have the *impression* of having the database to *itself*.

In order to explain how the concurrency control problem can be solved, several concepts must be introduced.

2.2.1 Conflict Serializability

The goal of this section is to define and explain the concept *conflict serializability*. But in order to do so, many other concepts must be introduced. The first concept to be introduced is *transaction history*.

Transaction History

A transaction *history*, also known as a *schedule*, is a collection of database operations done by one or more transactions on one or more data items. A possible transaction history that corresponds to *the lost update* example presented in section 2.1.1 could be as follows.

Example of a history, H_1. Let x denote the data item *count of available seats* on the actual flight. Assume that transaction t_1 is owned by Bob and that transaction t_2 is owned by Alice. Please note that "!" is used to denote comments.

1. t_1(b)
2. t_2(b)
3. t_1(r, x) ! x = 2
4. t_2(r, x) ! x = 2

5. t_1(w, x, 0) ! x = 0
6. t_2(w, x, 1) ! x = 1
7. t_1(c)
8. t_2(c)

Note that this is *not the only* transaction history that can represent the effect in a database caused by the execution from *the lost update* example. One could e.g. swap lines 3 and 4, and get the same effect.

Serial history. Histories represent executions, and to find out whether an execution is correct or not, its history can be *analyzed*.

When programs or transactions are assumed to be coded correctly, a history with *no* interleaving represents a *correct* execution of the transactions in it, and it is called a *serial* history. We can hence conclude that *serial histories are correct*.

Example of a serial history, H_2. Again, let x denote the data item *count of available seats* on the actual flight, let the transaction t_1 be owned by Bob, and let transaction t_2 be owned by Alice.

1. t_1(b)
2. t_1(r, x) ! x = 2
3. t_1(w, x, 0) ! x = 0
4. t_1(c)
5. t_2(b)
6. t_2(r, x) ! x = 0
7. t_2(a) ! No more reservations can be made since x = 0.

This history is serial and correct. Instead of doing an overbooking, Alice can tell her customer that there are no more seats available on the actual flight.

Conflict Equivalence

We have seen above that serial histories are correct. Next we want to find out how we can achieve non-serial histories that are guarantied to be correct. This is important, as mentioned in section 2.1, because interleaved executions of programs are often necessary due to performance reasons. We will soon see that *conflict serializable histories* are non-serial and correct, but first consider the two following definitions.

Conflict equivalent histories. Two histories, H_i and H_j, are *conflict equivalent* if:

1. they contain the same transactions and the same operations, and
2. operations *in conflict* have the *same* order in both histories.

Conflicting operations. Two operations are *in conflict* if they belong to *different* transactions, at least one of them is a *write*, and they access the *same* data item.

Conflict Serializability

Now we are able to define a *correct* and *non-serial* history, namely a history which is *conflict serializable*.

Conflict serializable history. A history is *conflict serializable* if and only if it is *conflict equivalent* to a *serial* history.

Example of a conflict serializable history, H_3. Again, let x denote the data item *count of available seats* on the actual flight, let the transaction t_1 be owned by Bob, and let the transaction t_2 be owned by Alice. Assume that there are *ten* seats available.

1. t_1(b)
2. t_1(r, x) ! x = 10
3. t_2(b)
4. t_1(w, x, 8) ! x = 8
5. t_2(r, x) ! x = 8
6. t_2(w, x, 7) ! x = 7
7. t_2(c)
8. t_1(c)

This history is conflict serializable because it is conflict equivalent to the serial history which first contains all the operations of t_1 and then all of t_2.

Remark. An other definition of equivalence is *view equivalence*. The consept *view serializability* is based on that definition. This approach is not common in practice and will just be mentioned here. More information about view equivalence can be found in e.g. (Bernstein et al., 1987, pages 38–41).

What is found? We have seen that a non-serial history is *correct* according to the definition of conflict equivalence, if and only if it is conflict equivalent to a serial history.

2.2.2 Which Histories are Conflict Serializable?

We have just seen that conflict serializable histories are correct. Next we will see that a conflict serializable history can be *recognized* by considering its *serialization graph.*

Serialization Graph

(Anfindsen, 1997, pages 8–9) writes:

> *A serialization graph (SG) for a transaction history H, is denoted SG(H). This is a directed graph whose nodes are the committed transactions of H, and it has an edge between all pairs of nodes where the two transactions in question have issued conflicting operations. The direction of the edges are in accordance with the sequence of the conflicting operations; from the former to the latter. In other words, an edge from t_i to t_j in SG(H) means that t_i has issued an operation that conflicts with and precedes some operation issued by t_j.*

Example of a Serialization Graph

Let the transactions t_1 and t_2 be as before, let the new transaction t_3 do the three operations t_3(b), t_3(r, y), and t_3(c), and let x and y be two data items. An execution of these transactions is represented in the following history, H_4.

1. t_1(b)
2. t_2(b)
3. t_1(r, x)
4. t_2(r, x)
5. t_3(b)
6. t_3(r, y) ! Note that y is different from x.
7. t_1(w, x)
8. t_2(w, x)
9. t_3(c)

10. t_1(c)

11. t_2(c)

H_4's corresponding *serialization graph*, SG(H_4), is shown in figure 2.1. It has three nodes, where each node represents a committed transaction.

There is a directed edge from t_1 to t_2 because the operations t_1(r, x) and t_2(w, x) conflict and because t_1(r, x) has been executed before t_2(w, x). There is also a directed edge from t_2 to t_1 because the operations t_2(r, x) and t_1(w, x) conflict and because t_2(r, x) has been executed before t_1(w, x). These are the only edges in the graph since t_3 has no conflicting operations neither with t_1 nor with t_2.

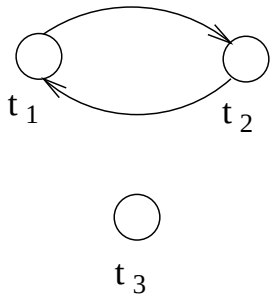

Figure 2.1: H_4's serialization graph, SG(H_4).

H_4 is *not* conflict serializable for the reasons explained next. Two requirements for a serial history which is conflict equvalent to H_4, are:

1. t_1 must precede t_2 because t_1(r, x) has been executed before t_2(w, x), and conflicting operations must have the same order in two conflict equivalent histories.

2. t_2 must precede t_1 because t_2(r, x) is executed before t_1(w, x), and conflicting operations must have the same order in two conflict equivalent histories.

But no serial history can satisfy both two requirements, so H_4 cannot be conflict serializable.

The two requirements can be seen in the serialization graph as a *cycle*. The first requirement is shown by a directed edge from t_1 to t_2, and the second by a directed edge from t_2 to t_1. This motivates the *serializability theorem* that will be presented next.

The Serializability Theorem

(Anfindsen, 1997, page 9) writes:

> *If t_i and t_j are involved in a cycle in SG(H), then t_i comes both before and after t_j in H, in which case H cannot possibly be equivalent with any serial history. This idea is formalized in the serializability theorem, also known as the fundamental theorem of serializability theory, which says that a history H is serializable if and only if SG(H) is acyclic (Bernstein et al., 1987).*

Which histories are conflict serializable? The histories that have *acyclic* serialization graphs are conflict serializable.

2.2.3 How can the DBMS Ensure Conflict Serializability?

If a DBMS can make sure that all histories have acyclic SGs, then the concurrency control problem is solved. This is because if all histories have acyclic SGs, then it is ensured that all histories are conflict serializable. When all histories are conflict serializable, they are correct, and so are the executions which they represent.

Three methods that can be used to ensure acyclic SGs are *serialization graph testing*, *timestamp ordering*, and *two phase locking* (2PL). Further information about the first two methods is given in e.g. (Bernstein et al., 1987, pages 114–128), while the *two phase locking* method will be considered next. The two phase locking method is widely used in practice. Throughout this thesis, locking will be the only concurrency control method to be considered.

Two Phase Locking (2PL)

(Anfindsen, 1997, page 9) writes that 2PL operates according to the following rules:

1. A transaction may not perform an operation on a data item unless it holds a lock corresponding to the operation in question on that data item.

2. A lock request from a transaction must be delayed or rejected by the scheduler if another transaction holds a conflicting lock on the data item in question.

3. A transaction may not aquire a new lock if it has released any of its old ones.

Explanation of the Two Phase Locking Protocol

(Anfindsen, 1997, page 9) writes:

> *Rules 1 and 2 prevent transactions from directly interfering with each other. Rule 3, the two phase rule, prevents cycles in the SG.*

The intuitive explanation of the latter goes as follows: When a transaction acquires a lock, that may establish an incoming edge to its node in the SG. An outgoing edge from a transaction's node in the SG can only be established if that transaction has released a lock. Thus, in order to create a cycle in the SG, some transaction must first release a lock and then later acquire a lock. Since this is prohibited by the two phase rule, 2PL schedulers will ensure acyclicity of SGs, and therefore ensure serializable histories. A formal proof for this was first given by (Eswaran et al., 1976). The result is very important, since 2PL is the most commonly used concurrency control algorithm in commercial systems. A thorough exposition and analysis of several variants of 2PL can be found in (Bernstein et al., 1987, pages 47–105).

Deadlocks

Deadlocks can occur when the 2PL method is used. For more information about deadlocks, see e.g. (Bernstein et al., 1987, pages 56–58).

The I in ACID is Achieved

By using 2PL, all histories are enforced to be conflict serializable and every transaction has the impression of having the database to itself. Then transactions have the I property of ACID.

2.2.4 Summing up

It has been shown how the concurrency control problem can be solved by enforcing 2PL. When 2PL is enforced, conflict serializable histories are ensured, and so is the isolation property of ACID.

2.3 The Recovery Problem

In this section we will consider the recovery problem. When a transaction has begun, a failure can occur *before* it has committed or *after* it has committed. If a failure occurs before the transaction has committed, then the transaction's effects in the database must be *rolled back* due to the A property of ACID. If a failure occurs after the transaction has committed and some or all of the effects are only saved to a log, then the transaction's effects must be permanently saved in the database, or equivalently *redone*, due to the D property of ACID.

We will not consider *how* recovery, or equivalently *roll back* and *redo*, can be done, but we will consider whether recovery is *possible* to do or not. In order to find out, histories will be analysed.

2.3.1 Recoverable (RC) Histories

It is assumed in the following that the 2PL protocol is enforced in order to achieve conflict serializable histories. Unfortunately, recovery is not always possible when this protocol is enforced, but it *will be* when an *extra* requirement is added. These statements are explained below, but first it is necessary to define what is meant by "*the transaction t_i has read from the transaction t_j*".

What is meant by "t_i has read from t_j"?

If the transaction t_i has read an item previously written by the transaction t_j, then t_i *has read from* t_j.

Example of a Non-Recoverable History, H_5

Again, let x denote the data item *count of available seats* on the actual flight, and let the transactions t_1 and t_2 be as before. Assume that there are *ten* seats available.

A possible execution of these transactions is represented in the following history, H_5, where the 2PL protocol is followed.

Please note that when a transaction t_i gets a read lock on an item x, it will be denoted t_i(rl, x), and similarly t_i(wl, x) when it gets a write lock.

1. t_1(b)
2. t_2(b)
3. t_1(rl, x)
4. t_1(r, x) ! x = 10
5. t_1(wl, x)
6. t_1(w, x, 8) ! x = 8
7. t_1 releases its locks.
8. t_2(rl, x)
9. t_2(r, x) ! x = 8
10. t_2(wl, x)
11. t_2(w, x, 7) ! x = 7
12. t_2 releases its locks.
13. t_2(c)
14. t_1(a) ! A system failure occurs.

When the failure occurs, t_1 is aborted and must be rolled back due to the A of ACID. The transaction t_2 has read from t_1, and consequently its results are based on those of t_1. Then t_2 must be rolled back too, but it is impossible to roll back a committed transaction due to the D of ACID. Thus H_5 is not recoverable.

How do we get Recoverable Histories?

If the 2PL protocol is enforced and if it is required that a transaction *cannot commit before* all transactions from which is has read have committed, then all histories will be *conflict serializable* and *recoverable*.

The situation shown in H_5 will not happen if the extra requirement is enforced because a commit attempt from t_2 before t_1 will be disallowed.

The ACID Properties are Achieved

The C property of ACID is achieved by assumtion, as we saw in section 2.1, the I property of ACID is achieved when the 2PL protocol is enforced, as we saw in section 2.2, and the A and D properties are achieved when it in addition is required that a transaction must commit after all transactions from which it has read have committed.

In other words, the ACID properties are achieved when histories are conflict serializable and recoverable.

2.3.2 Avoiding Cascading Aborts (ACA)

There is an undesirable phenomenon called *cascading aborts*. An example of cascading aborts follows.

Assume that two transactions t_i and t_j have read from a transaction t_k and that t_k has read from a transaction t_l. Then if t_l aborts, t_k must be aborted since it has read from t_l. Also, because t_j and t_i have read from t_k, they must be aborted.

If in addition other transactions have read from t_i and t_j, then they must be aborted and so on, resulting in a *cascade* of aborts.

Cascading aborts can be avoided if transactions only are allowed to read from *committed* transactions. This is because when all transactions which a transaction have read from have committed, it is impossible for a transaction to read uncommitted data that possibly could cause the transaction to be aborted in the future.

Note that when cascading aborts are avoided, histories will also be recoverable.

If the 2PL protocol and this new requirement are enforced, then all histories will be *conflict serializable*, *recoverable*, and *ACA*.

2.3.3 Strict (ST) Histories

Recovery becomes simpler if yet another requirement is enforced. This will be explained below.

A History Which is not Strict

The following history is conflict serializable and ACA, but *not* strict. Assume that the transactions t_4 and t_5 update some data item z without reading it first. A possible execution of the two transactions is represented by the following history, H_6.

1. t_4(b)
2. t_5(b)
3. t_4(wl, z)
4. t_4(w, z)
5. t_4 releases its lock.
6. t_5(wl, z)
7. t_5(w, z)
8. t_5 releases its locks.
9. t_5(c)
10. t_4(a) ! A system failure occurs.

First t_4 writes z, a little later t_5 writes z, next t_5 commits, and then t_4 aborts. Since t_4 was aborted it must be rolled back, and it would have been easy if the value of z written by t_4 just could have been replaced by z's *before image*. If no committed transaction had written a new value of z, then z's before image *could* have been restored. But since t_5 has written z *after* t_4 *and* committed, a restoring of the before image in question would have overwritten the result of t_5 which is supposed to be durable.

This kind of problems are manageable but they require advanced algorithms in order to do recovery. Matters are simplyfied if a transaction is restricted to only write data items that previously have been written by *committed* transactions. Then the situation in H_6 could not have happenend because t_5 would have been *disallowed* to write access z until t_4 had committed or aborted. In practice t_5 would have been disallowed to write access z until t_4 had committed or aborted because t_4 would have *held its write lock* until the end of transaction.

Note that when a history is strict, then it will also be ACA. This is because when all write locks of a transaction are held until the transaction commits or aborts, then it will be impossible to read data written by that transaction until it has ended since write locks conflict with all other locks. Consequently, an arbitrary transaction will only be able to read from committed transactions, and it will therefore be ACA.

So when the 2PL protocol and this new requirement are enforced, all histories are guaranteed to be *conflict serializable*, *recoverable*, *ACA*, and *ST*.

2.3.4 Rigorous (RG) Histories

(Anfindsen, 1997, page 11) writes:

> *Thus with a 2PL protocol that retains all write locks until the end of transactions, we can guarantee both strict and conflict serializable histories. That's fine, but how does the scheduler know when it's safe to start releasing read locks? Unfortunatly, it doesn't. ... Thus, locking schedulers that want to ensure both strict and conflict serializable histories, must actually go one step further than indicated above, they must retain* all *locks, not just write locks, until the transaction terminates.*

A strict history with the additional property that a data item that has been read by one transaction cannot be overwritten by any other transaction until the former has committed is known as rigorous (Anfindsen, 1997, page 11).

So when the 2PL protocol, the ST requirement, and this new requirement are enforced, all histories are guaranteed to be *conflict serializable*, *recoverable*, *ACA*, *ST*, and *RG*.

2.3.5 Summing up

In this section it has been considered whether recovery is *possible* to do or not, and histories were analyzed in order to find out.

When the 2PL protocol, the ST requirement, and the *RG* requirement are enforced, all histories are guaranteed to be *conflict serializable*, *recoverable*, *ACA*, *ST*, and *RG*. The ACID properties are achieved when histories are conflict serializable and recoverable.

More information about recovery can be found in e.g. (Bernstein et al., 1987).

2.4 Summary

Transaction theory was developed in order to ensure users a correct view of databases. Errors in databases are due to concurrency or to failure, given

that programs are coded correctly.

Transaction theory is about solving the concurrency control- and the recovery problem.

If the ACID properties of transactions are ensured, then databases will be correct.

The I of ACID is achieved when the concurrency control problem is solved. We saw how this problem could be solved by enforcing the 2PL protocol in order to achieve conflict serializable histories.

The A and D of ACID are achieved when it is required that a transaction can only commit after all the transactions from which it has read, and the resulting histories are known as recoverable.

The C of ACID is achieved by assuming that transactions are coded correctly.

It is pleasant to have histories that avoid cascading aborts and are strict. In practice one usually goes one step further and makes all histories rigorous.

Chapter 3

Advanced Transactions

In this chapter we will consider *advanced* database applications that have different requirements to transaction processing than classical or traditional applications.

3.1 Transaction Model

In order to meet the requirements of advanced database applications, research has led to many advanced *transaction model* proposals.

But what is a transaction model? (Anfindsen, 1997, page 24) uses the following definition:

> *A transaction model is a specification of allowable and mandatory behaviour for transactions as well as their structure.*

For example a transaction model can specify that transactions must have a flat structure or a nested structure. Further it can specify behavior like what type of concurrency- and recovery scheme transactions must follow.

The advanced transaction model Apotram presented by Ole Jørgen Anfindsen in (Anfindsen, 1997) will be introduced in chapter 4. Descriptions of other advanced transaction models can be found in (Elmagarmid, 1992).

3.2 Classical and Advanced Transactions

A motivation for why advanced transaction models are needed will be given in this section.

Advanced database applications have different requirements to database systems than classical applications. Among these are new requirements for *data models* and *transaction processing*. Here we will focus on some of the new requirements for *transaction processing*. In (Elmagarmid et al., 1992, page 34) the following is written:

> *Recently there has been widespread use of databases in advanced (non-traditional) applications. It has been found that the traditional transaction concept has limited applicability in many of these advanced applications. For example, in CAD/CAM, office automation, publication environments and software development environments, transactions are usually very complex, need to access many data items and reside in the system for a long duration. Transactions of this kind are usually called long-lived transactions.*

We will look at some of the differences between classical and advanced transactions next.

In chapter 2 we discussed *classical* transaction theory and the *ACID properties*. Those properties are very important to many applications in order to give a correct view of databases. Common for these applications is that the transactions are *short lived*, *programmed*, and *total isolation from other transactions is essential*.

For many other database applications the situation is different. Often advanced transactions like design transactions are *interactive*, *long lived*, and *cooperative*.

An *interactive* transaction is taken control of by a *human*. The human decides the actions of the interactive transaction along the way of his or her design work. An interactive transaction can start when the human chooses *begin transaction* from a menu in a graphical user interface (GUI) and does not end until he or she chooses *commit* or *abort* from a menu in the GUI (seen in absence of failures). Between the *begin transaction* command and the termination of the transaction, the human designs. This type of transaction is the opposite to the classical non-interactive transaction that can be written as a program application.

Interactive transactions are of *long duration* because the human activity involved may take hours, days, or even a longer period (Silberschatz et al., 1997, page 679).

Often long lived transactions need to *cooperate* with other long lived transactions. We will look at two types of cooperation next.

3.2.1 Cooperation

Data Sharing

The first type of cooperation we will consider is *data sharing*. This means that transactions need to look at each others *uncommitted* modified data in order to let designers do their jobs. To get an intuitive understanding of what this means, we will take a look at the following example taken from (Anfindsen, 1997, pages 30–31):

> *Consider the designers Bob and Alice working on a new aeroplane. Let's say Alice is working on the landing gear and Bob is working on the hydraulic system, with both sets of objects stored in a design database. Both Bob and Alice execute transactions that last for days or weeks. Given that the landing gear and hydraulic system must be properly interfaced, Bob and Alice will both repeatedly need to look at the design objects of the other. This will cause cyclic write-read dependencies to be established between their transactions, which is incompatible with serializability.*

In addition (Elmagarmid, 1992, pages 34–35) writes the following about some of the limitations of CAD transactions in a traditional transaction processing environment:

> *It does not allow much cooperation between activities. A CAD project is usually performed jointly by a group of people; each person is responsible for a part of the design project. People of the same group have to cooperate in order to achieve a good design. One means of cooperating is by exchanging information through shared data items. To do this, people (or their corresponding transactions) need to access the shared data items alternately. The traditional transaction concept severely restricts this type of cooperation by requiring the isolation of uncommited transaction results.*

So while classical transactions need *total isolation* among transactions in order to get correct results, many advanced transactions need to *share data* with other transactions in order to let designers do their jobs.

Teamwork

The second type of cooperation we will consider, I call *teamwork*. By teamwork I mean that more than one transaction modify a certain object before the object is committed to the outside world. The following example given in (Anfindsen, 1997, pages 48–49) will make this more concrete:

> *Consider the process of designing a new aeroplane. Presumably, there will be several teams of engineers working on various parts of the aeroplane such as the wings, the hydraulic system, the landing gear, etc. Let's say that Alice, Bob, and Catherine have been assigned the task of designing the landing gear, that Alice has the main responsibility for the front wheels, that Bob and Catherine work together on the rear wheels, and that Alice is the leader of this team. As long as Alice, Bob, and Catherine are*

> *working on disjunct sets of objects, ordinary concurrency control would be fine. But in the long run that would be unsatisfactory. For example, Bob might be responsible for interfacing all landing gear objects with the electric system, Alice might be responsible for interfacing all landing gear objects with the hydraulic system, and both of these systems are being designed by other teams. This makes controlled but non-serializable interaction patterns desirable.*

So while classical transactions need *total isolation* among transactions in order to get correct results, many advanced transactions need to do *teamwork*, or equivalently update objects in an alternately fashion, before the results are committed.

3.2.2 The ACID Properties

The ACID properties are too restrictive for advanced transactions as described above, or actually it is only the *isolation* property of ACID that is too restrictive. One still wants the atomicity, consistency, and durability properties (Anfindsen, 1997, pages 6–7).

We will next concider more details about the I property and advanced transactions.

3.3 The Isolation Property

As we saw in section 2.2 the usual way to achieve the I property of ACID is to ensure the correctness criterion *conflict serializability (CSR)* by enforcing the 2PL protocol. Please note that the only concurrency control implementation technique to be considered in this thesis is locking.

Also note that it is not the *implementation technique* that is to restrictive for advanced transactions, but *the I property of ACID*. If other concurrency control implementation techniques were used to achieve isolation, then the I property of ACID would still be too restrictive for advanced transactions (Kaiser, 1995, page 411).

In the *data sharing* example, the desired access pattern would cause a cycle in the serialization graph if CSR was the correctness criterion, and therefore according to CSR, the resulting history would be incorrect. A correctness criterion more lenient than CSR is prefered by advanced transactions. They want read-write and write-read conflicts to be handled in a more lenient but still controlled way. A way to achieve this will be presented in section 4.2 about *conditional conflict serializability (CCSR)*, one of the correctness criteria defined in (Anfindsen, 1997).

In the *teamwork* example, more than one write lock would be needed on an object in order to do teamwork. A controlled way to achieve this will be presented in section 4.3 about *nested conflict serializability (NCSR)*.

3.4 What about Classical Recovery Management?

If a transaction lasts for many weeks and a system failure occurs a few minutes before the transaction is ready to commit, then the work done will be rolled back due to the A property of ACID. Obviously this is not acceptable. Instead it is desirable for the user to make *savepoints* along the design way. Then if a system failure occurs, the roll back can be limited to a previously taken savepoint.

The A and D properties of ACID are still satisfied when savepoints are used (Anfindsen, 1997, page 5–6).

3.5 Summary

In this chapter it has been motivated that the I property of ACID and consequently the correctness criterion CSR is inapproperiate for advanced transactions, while the A and D properties of ACID are still desirable when savepoints are supported.

As in the classical case one assumes C. This means that application programmers or owners of interactive transactions are responsible for correct logic of their transactions. Then in absence of controlled cooperation and failures, the transactions are assumed to take the database from one consistent state to a new consistent state.

The I property of ACID is too restrictive for advanced transactions because of their need to cooperate either by sharing data and/or by doing teamwork.

More information about this topic is given in e.g. (Kaiser and Pu, 1992), (Weikum and Schek, 1992), (Unland and Schlageter, 1992), and (Nodine et al., 1992).

Chapter 4

Apotram

4.1 Introduction

Apotram was first presented in the PhD thesis of Ole Jørgen Anfindsen (Anfindsen, 1997). A prototype implementation and a proof-of-concept demo application of Apotram have been made by Anfindsen and the Persistent Java team as a part of the Forest Project at SunLabs. More about this can be found at <http://www.apotram.com>.

As discussed in chapter 3, isolation (and consequently conflict serializability) is too restrictive for advanced transactions with cooperation requirements.

Apotram *extends* the classical transaction model discussed in chapter 2 in order to meet advanced transactions' cooperation requirements. It is an *application-oriented transaction model*, the goal of which is to specify a way in which applications can *dynamically influence* the way transaction management is carried out (Anfindsen, 1997, page xi).

The *default* transactional behavior of Apotram is *conflict serializability*. However, if a more lenient style of concurrency control is desirable, then two new correctness criteria are available.

First the correctness criterion *conditional conflict serializability (CCSR)* will be introduced in section 4.2. CCSR lets read-write and write-read conflicts to be customized in order to allow *controlled* data sharing.

Second the correctness criterion *nested conflict serializability (NCSR)* will be introduced in section 4.3. NCSR handles write-write conflicts in order to allow *teamwork* in a *controlled* way.

Finally, the two new correctness criteria will be *combined* resulting in the correctness criterion *nested conditional conflict serializability (NCCSR)*, which is the correctness criterion of Apotram. This correctness criterion allows both kinds of cooperation discussed in section 3.2.1, and is presented in section 4.4.

The rest of this chapter is based on (Anfindsen, 1997).

4.2 Conditional Conflict Serializability (CCSR)

In this section we will see how the correctness criterion CCSR enables read-write and write-read conflicts to be *customized*. By this it is meant that users (for example owners of interactive transactions) can influence the way concurrency control is carried out. They can to some degree define which operations conflict and which commute *dynamically*. In practice this correctness criterion enables *data sharing*.

Note that the definitions that follow in this section are from (Anfindsen, 1997, page 29–30).

4.2.1 The new Definitions

In order to define CCSR we need a new notion of conflict, namely *conditional conflict*.

Conditional Conflict

The *parameterized* read mode r(A) and the *parameterized* write mode w(B) conflict unless B $\subseteq$ A.

Two write operations that belong to *different* transactions and access the *same* data item are still considered as *conflicts*.

Parameterized Access Modes

The new definition of conflict contains the notion of *parameterized* read- and write mode. We will now see what is intuitively meant by that. (Anfindsen, 1997, page 31) writes the following:

> *The basic idea of parameterized read and write modes is that users should be able to specify when reading and writing should be incompatible. In other words, the standard notion that read and write modes are mutually incompatible is reduced to a default which transactions may override by proper use of parameters.*

(Anfindsen, 1997, page 31) also writes the following:

> *The motivation behind CCSR is twofold:*
>
> 1. *It enables application programmers to customize the notion of conflict, and*
> 2. *It enables them to communicate to each other the quality of uncommitted data.*
>
> *The access mode parameters are the key to both of these aspects.*

Each transaction can have a *read parameter set (rps)* and a *write parameter set (wps)*, where the parameters contained in the parameter sets are chosen from a parameter domain D containing all the available parameters in the system. Examples of two parameters are *Incomplete Draft (ID)* and *Complete Draft (CD)*.

Example. Recall the *data sharing* example from section 3.2.1 where Alice and Bob were working on a new areoplane. Alice was working on the landing gear and Bob was working on the hydraulic system. Let us make a simplification and say that the hydraulic data is contained in the object H.

Now if Bob is willing to share the data object H with others, then he can use a wps. Let his wps for example be B_{Bob}={ID}.

As we saw above, the read mode r(A) and the write mode w(B) *conditionally conflict* unless B $\subseteq$ A. In other words, r(A) and w(B) are compatible if and only if B $\subseteq$ A.

So by using the wps in question, Bob tells the system that a read operation r(A), which uses an rps A, such that A $\supseteq$ {ID}, does *not* conditionally conflict with his write operation (w(B_{Bob}), H), and he tells other possibly existing transactions that hold read locks on the object H about the hydraulic data's incomplete state.

So if Alice for example uses an rps A_{Alice}={ID, CD}, then the read operation (r(A_{Alice}), H) will not conditionally conflict with Bob's write operation (w(B_{Bob}), H). By using an rps, Alice tells the system that she is willing to read uncommitted data belonging to transactions which have wpss that are *equal to or subsets of* A_{Alice}.

Note that if the classical definition of conflict was used, then Bob's write operation (w, H) and Alice's read operation (r, H) would conflict. But instead of considering classical read-write and write-read conflicts as conflicts, read-write and write-read "conflicts" can be thought of as *conditional conflicts* where transaction owners specify the conditions by using parameter sets. However, the classical notion of conflict will still be valid if no parameters are used. This means that the standard notion of conflict is reduced to a *default*.

Several parameterized read- and write modes. We will now consider a few examples of parameterized read- and write modes.

Let the domain D contain the elements u1, u2, and u3. Let the sets A and B be such that A $\subseteq$ D and B $\subseteq$ D. As we saw above r(A) and w(B) are compatible if and only if B $\subseteq$ A. Then it would be the case that:

- r(u1) and w(u1) are compatible.
- r(u1) and w(u2) are incompatible.
- r(u1, u2) and w(u2) are compatible.
- r(u2) and w(u2, u3) are incompatible.

The unparameterized read moder and the read mode r(∅) should be considered the same read mode. Similarly should the unparameterized write mode w and w(*) be considered as the same write mode where * denotes an arbitrary superset of D. Thus, according to the conditional conflict definition stating that r(A) and w(B) are compatible if and only if B ⊆ A, r(∅) will be incompatible with all write modes and w(*) will be incompatible with all read modes.

Note that the parameter domain D is user defined and that it can contain few or many parameter values, depending on applications' need. It must be known to the users what the actual meaning of each parameter is.

Conditional Conflict Equivalence

Based on the new definition of conflict, *conditional* conflict equivalent histories are defined next.

Conditional equivalent histories. A history H_i is said to be conditional conflict equivalent to an other history H_j if:

1. they contain the same transactions and the same operations; and
2. conditionally conflicting operations of non-aborted transactions are ordered in the same way in both histories.

Conditional Conflict Serializability (CCSR)

We are now able to define a *conflict serializable history* which is *correct* according to the correctness criterion *CCSR*.

Conditional conflict serializable history. A history is defined as *conditional conflict serializable* (CCSR) if and only if it is conditionally conflict equivalent to a serial history.

What is found? According to the new concept of *conditional* conflict and *conditional* conflict equivalent histories, a history is correct if and only if it is conditionally conflict equivalent to a serial history. Conflicts are conditional because they depend on transactions' parameter sets.

4.2.2 Which Histories are CCSR?

We have just seen that conditional conflict serializable histories are correct. Next we will see that a conditional conflict serializable history can be recognized by considering its *conditional conflict serialization graph*. The definition of conditional conflict serialization graph is similar to the definition of a serialization graph given in section 2.2.

Conditional Conflict Serialization Graph

A *conditional conflict serialization graph (CCSG)* for a trasaction history H is denoted CCSG(H). CCSG(H) is a directed graph whose nodes are the committed transactions of H, and it has an edge between all pairs of nodes where the two transactions in question have issued *conditionally conflicting* operations. The direction of the edges are in accordance with the sequence of the *conditionally conflicting* operations; from the former to the latter. In other words, an edge from t_j to t_k in CCSG(H) means that t_j has issued an operation that is in conditional conflict with and precedes some operation issued by t_k.

The Generalized Serializability Theorem

A history H is CCSR if and only if the *conditional conflict serialization graph* CCSG(H) is acyclic.

Proof. The proof of the generalized serializability theorem will not be given here, but it is given in (Anfindsen, 1997, page 30).

Example of a CCSG

Once again consider Alice and Bob who is working on a new aeroplane. Alice works on the landing gear and Bob works on the hydraulic system. Let us make a simplification and say that the hydraulic data is contained in the object H and that the landing gear data is contained the object L.

The following scenario is also shown in the history H_7 below. Bob begins, reads H, does some work on the object (not shown in the history), and then writes H back to the database in w(ID) mode. Alice begins and reads L. She wants to see what Bob has done, so she reads H in r(ID, CD) mode. She does some work on her object L (not shown in the history) and then writes L back to the database in w(ID) mode.

Bob wants to see Alice's work, so he reads L in r(ID, CD) mode. Then he finishes his work on H and writes it back to the database in w(CD) mode. Finally, Alice reads H in r(ID, CD) mode, adjusts her work, and writes L back to the database in w(CD) mode. The history H_7 is shown next.

1. t_B(b)
2. t_B(r, H)
3. t_B(w(ID), H)
4. t_A(b)
5. t_A(r, L)

6. t_A(r(ID,CD), H)
7. t_A(w(ID), L)
8. t_B(r(ID,CD), L)
9. t_B(w(CD), H)
10. t_A(r(ID,CD), H)
11. t_A(w(CD), L)
12. t_B(c)
13. t_A(c)

The graph CCSG(H_7) has no edges because there are no conflicting operations in it. Then CCSG(H_7) is acyclic, and consequently H_7 is CCSR (and correct).

If t_A had read H in r(CD) mode instead of r(ID,CD) mode on line 6, then its read operation would conflict with the operation t_B(w(ID), H) on line 3. This would cause an edge to occur from t_B to t_A in the history's CCSG since t_B's actual operation preceedes the one of t_A. If in addition t_B had read L in r(CD) mode instead of r(ID,CD) mode on line 8, then its read operation would conflict with t_A(w(ID), L) on line 7. This would cause an edge to occur from t_A to t_B in the history's CCSG since t_A's actual operation preceedes the one of t_B. These two edges would cause a cycle in the history's CCSG, and consequently the history would not be CCSR.

Remark. Consider the operations on line 3, 9, and 10 in H_7. Assume now that the last operation is changed to t_A(r(CD), H). Then this operation conflicts with the write operation on line 3. But since the write operation on line 9 writes the same object as the one on line 3 in a new write mode, this last write mode is the one to be compared with the actual read mode. So this situation would not lead to a cycle in the CCSG of the actual history since the write mode w(CD) and the read mode r(CD) do not conditionally conflict.

Equivalence to a serial history. H_7, as it was shown above, is conditionally conflict equivalent to e.g. the serial history containing all operations of t_B before all those of t_A.

First it can seem strange that a conditional conflict serializable history like H_7 *can* be equivalent to a serial history since there is information exchanged between the transactions in the non-serial case that is not exchanged in the serial case.

But when information is exchanged, it is considered by users and they *judge* the data and decide what to do next according to the information they get. Because of this *judgement* done by users, the execution can be considered correct even though uncommitted data is exchanged.

While users judge information given in parameter sets, the system considers an execution as correct if the execution's CCSG is acyclic.

So the responsibility of the concurrency control is shared between the system and the users. The system controls that histories are conditionally conflict equivalent to serial histories, while users are responsible of interpreting the information given in parameter sets in a proper way.

4.2.3 How can CCSR be Enforced?

A history is CCSR if its CCSG is acyclic, we will see next how a DBMS can make sure that all histories have acyclic CCSGs.

Parameterized Two Phase Locking (2PPL)

The 2PPL rules, similar to the 2PL rules presented in section 2.2.3, follow:

1. A transaction may not perform an operation on a data item unless it holds a lock corresponding to the operation in question on that data item.

2. A lock request from a transaction must be delayed or rejected by the scheduler if another transaction holds a *conditionally* conflicting lock on the data item in question.

3. A transaction may not aquire a new lock if it has released any of its old ones.

The only difference to the rules of regular 2PL is that we now consider *conditional* conflicts instead of classical conflicts. If a transaction shall read an item x and it has an rps A_t, then an r(A_t) lock must be held on item x before the transaction can access it.

The implementation technique 2PPL ensures histories with acyclic CCSGs for the same reasons as 2PL ensures histories with acyclic SGs explained in section 2.2.3.

4.2.4 CCSR and Recovery

As mentioned in section 3.2, the A and D propertites of ACID are still desirable. The following shows how these properties can be achieved given that 2PPL is the concurrency control mechanism enforced. If more details about CCSR and recovery are desirable, please see in (Anfindsen, 1997, pages 29–43).

What is meant by "t_i has read from t_j in a conflicting mode?"

Assume that we have two transactions t_i and t_j and that t_j wrote an item x *the last time* in a w(B) mode. Then if t_i has read x after the last time t_j wrote it *in an incompatible* read mode r(A), such that (A $\not\supseteq$ B), then we say that t_i has read from t_j in a conflicting mode.

Recoverable Histories modulo CCSR

A history is CCSR and recoverable *modulo* CCSR, denoted RC(CCSR), if each transaction commits after all transactions from which it has read in a *conflicting* mode have committed.

Avoiding Cascading Aborts modulo CCSR

If histories that are CCSR and avoid cascading aborts *modulo* CCSR, denoted ACA(CCSR), are desirable, then an arbitrary transaction t_i cannot read data in a conflicting mode from an arbitrary transaction t_j before t_j has committed. Note that histories which are ACA(CCSR) are also RC(CCSR).

Strict Histories modulo CCSR

If one also wants histories that are CCSR and strict *modulo* CCSR, denoted ST(CCSR), then an arbitrary transaction t_i should not be allowed to write data items written by another arbitrary transaction t_j until t_j has committed. This means that t_j must keep all its write locks until it has committed. Note that ST(CCSR) histories are also ACA(CCSR).

Rigorous Histories modulo CCSR

Since its hard to predict when a transaction stops aquiring locks, all read locks must be held until commit too, and the resulting histories are CCSR and rigorous *modulo* CCSR denoted RG(CCSR). Note that RG(CCSR) histories are also ST(CCSR).

4.2.5 Summing up

We have seen the new correctness criterion CCSR where data sharing is possible when parameterized access modes are used instead of regular read and write modes.

We have also seen that this correctness criterion can be enforced by using 2PPL. In addition we have seen how the A and D properties of ACID can be maintained even if we use CCSR.

The I of ACID is replaced by controlled data sharing.

4.3 Nested Conflict Serializability (NCSR)

We will now consider the second correctness criterion of Apotram. It handles write-write conflicts in a controlled way in order to deal with *teamwork*. This correctness criterion is called nested conflict serializability (NCSR).

The feature of Apotram that allows multiple users to update a given data item in a controlled manner is called *nested databases* (Anfindsen, 1997, page 45). However, before nested databases are presented, it is necessary to introduce the concept of *nested transactions*.

4.3.1 Nested Transactions

In (Unland and Schlageter, 1992, pages 407–408) the following is written:

> *The idea of nested transactions seems to have originated with (Davies, 1973) some time ago. ... (Reed, 1978) presented the first comprehensive design of a nested transaction system. This design uses timestamps for synchronization. It was not before 1981 and the work of (Moss, 1981) that nested transactions attracted greater attention in the database community.*

The nested transactions of Moss use locks for synchronization (Anfindsen, 1997, page 23).

The traditional transactions described in chapter 2 have a *flat* structure while the structure in a nested transaction is *hierarchial* or equivalently *nested*. A nested transaction has a top-level transaction which again has one or more subtransactions. Each of these subtransactions can again have one or more subtransactions and so on, to an arbitrary depth. See the illustration of a nested transaction in figure 4.1.

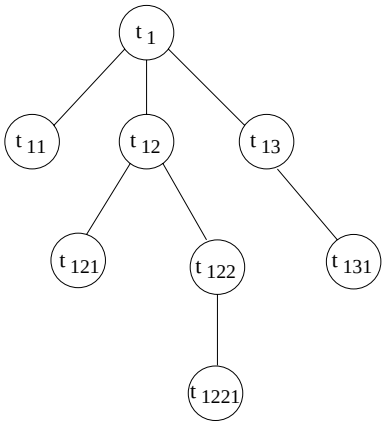

Figure 4.1: A Nested Transaction.

Terminology

Some of the basic terminology for nested transactions follows. The top-level transaction is also called a *root* transaction and the transaction hierarchy is also called a *tree*. Each subtransaction on the level immediately below a (sub)transaction are the (sub)transaction's *children* and the (sub)transaction is these children's parent. The children of a specific *parent* are called *siblings*. A subtransaction is either flat or nested. If it is flat, then it is called a *leaf* while a *nested* subtransaction is called a *subhierarchy* or *subtree*.

Example of a Nested Transaction

Consider figure 4.1. t_1 is the root and the parent of the subtransactions t_{11}, t_{12}, and t_{13}. These three subtransactions are t_1's children and they are siblings. t_{12} is the parent of t_{121} and t_{122}. All the transactions in the hierarchy or tree are subtransactions except t_1. t_{12}, t_{13}, and t_{122} are subtrees (or *nested* subtransactions) while t_{11}, t_{121}, t_{1221}, and t_{131} are leaves (or *flat* subtransactions). Note that the leaves in a transaction tree do not need to have the same distance from the root.

Rules of a Nested Transaction

Some rules of nested transactions follow. The *commit rule* of Moss' approach (Gray and Reuter, 1993, pages 195–197) *states* that a subtransaction must commit to its *parent* instead of to the database while the top-level transaction commits to the *database*. This means that subtransactions does *not* have the D property of ACID. Only when the top-level transaction commits to the database, the work done by the subtransactions in the nested transaction becomes durable and visible to the outside world.

So if a transaction in a transaction hierarchy aborts and is rolled back, then all its subtransactions will be rolled back too even though they have committed to their parent.

When a subtransaction commits to its parent, the work may be visible to siblings causing *intratransaction parallelism*.

Some advantages of nested transactions are the achievement of *modularity*, *local recovery*, and *intratransaction parallelism*.

If subtransactions commit to their parents, then nested transactions are called *closed* nested transactions.

Travel Planning

An often used example of nested transactions is the travel planning example where the tasks of a transaction are e.g. to reserve a seat on a flight, a room on a hotel, and a rental car in the destination town. When nested

transactions are used, the actual transaction can be divided into the three subtransactions, t_{flight}, t_{hotel}, and t_{car}, getting a natural modularization. Then if the flight and hotel reservations are successfully done but the car reservation causes a local roll back of t_{car}, then the two successful transactions do not have to be rolled back. The transaction t_{car} can be restarted and when all three subtransactions have committed successfully to their parent (here the root), then the root can commit to the database.

Nested Transactions as a Basis for Other Transaction Models

Nested transactions are used as a *basis* for many advanced transaction models like the *Tool Kit Approach for Transaction Management* presented in (Unland and Schlageter, 1992), the *Cooperative Transaction Model for Design Databases* presented in (Nodine et al., 1992), and the *Multilevel Transactions and Open Nested Transactions* presented in (Weikum and Schek, 1992).

For more information about nested transactions, see e.g. (Gray and Reuter, 1993, pages 195–200) and (Anfindsen, 1997, pages 23–27).

4.3.2 Nested Databases

Sphere of Control

Nested databases is a concept based on that of *sphere of control (SOC)* (See (Gray and Reuter, 1993, pages 174–180) and (Anfindsen, 1997, pages 45–47)). One can think of SOC as a general concept with the meaning indicated by its name. Some SOC examples follow. A database is a SOC that the DBMS is in control of, an interactive transaction is a SOC that its owner is in control of, and a set of data locks belonging to a transaction is a SOC that the transaction is in control of. An illustration of the examples is given in figure 4.2.

In the following we will consider transaction SOCs and data SOCs, we will assume that transaction nesting is supported even though it is not strictly necessary (Anfindsen, 1997, page 45), and still, locking is the only concurrency implementation technique that will be considered.

Nested Transactions and SOCs

Let us now take a look at nested transactions and SOCs. When a nested transaction is executing, it contains a hierarchy of (sub)transaction SOCs where each (sub)transaction SOC may contain a data SOC. A data SOC may be divided into read SOCs (RSOCs) and a write SOCs (WSOCs). Subtransactions begin and terminate dynamically so the hierarchy is dynamic as well.

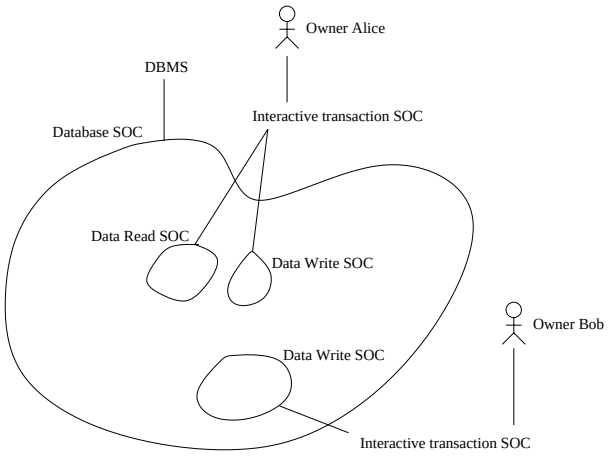

Figure 4.2: Sphere of control examples.

Recall from the description of nested transactions that the commit procedures of subtransactions in a closed nested transaction is different from those of a top-level transaction (in a nested transaction). Subtransactions must commit to their parents, and their work results are only durable when the top-level transaction of the nested transaction commits. When a subtransaction commits, it hands its WSOCs to its parent while its RSOCs cease to exist. When a top-level transaction commits, both its WSOCs and RSOCs will cease to exist.

Nested Databases

Now when the concept of SOC is introduced, the concept of nested databases can be explained. (Anfindsen, 1997, page 46) writes:

> *A crucial observation at this point is that a WSOC may be regarded as a special case of a database. Both are SOCs enclosing sets of data items, and both are used to control who is allowed to manipulate the enclosed data. One could say that a WSOC establishes a single user database where no transaction but the WSOC holder has access, and that this is why no concurrency control is necessary inside a WSOC. The idea upon which [the correctness criterion NCSR] is built, is to allow a WSOC to be dynamically converted to a database SOC at the will of the owning transaction. In doing so, the transaction in question creates a* subdatabase*, and can decide exactly which transactions should be allowed to access that subdatabase, and in which access modes. The subdatabase can be converted back to a WSOC when no transactions are active inside it, and the owning transaction*

has control over if and when the contents of the subdatabase are committed to the enclosing database.

Example. Assume that Alice is the owner of an interactive transaction t_A. She has a WSOC (and possibly an RSOC which we ignore in this example).

Now if she wants Bob and Catherine to do some work for her, then she can simply convert the WSOC to a subdatabase and mark that Bob and Catherine are allowed to access the subdatabase in e.g. write modes. Then Bob and Catherine can start transactions inside the subdatabase. If they modify objects in the subdatabase, then WSOCs belonging to Bob and Catherine will appear inside of it.

Further assume that Denise, who has not access to the subdatabase, accesses some objects outside the subdatabase. Consequently, she has only transactions outside the subdatabase. Bob, who is in control of a nested transaction, has subtransactions outside the subdatabase as well as inside it. See figure 4.3 for an illustration of the scenario.

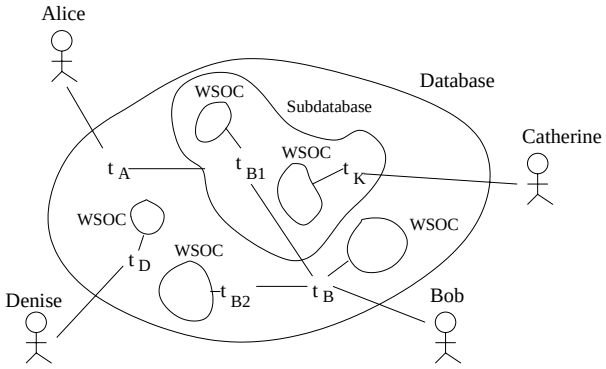

Figure 4.3: Subdatabase scenario.

Further Nesting

Subdatabases can further be nested. Any WSOC may at any time be converted to a subdatabase. Creation of subdatabases may continue to arbitrary depths (subject only to implementation limits).

Generalized Commit Rule

In an environment with nested databases, a new generalized commit rule is necessary.

If a *subtransaction* is inside a subdatabase, then it must commit its WSOCs to that subdatabase. Otherwise it should commit its WSOCs to its parent.

For example in figure 4.3 the subtransaction t_{B1} must commit its WSOC to t_A's subdatabase while t_{B2} must commit to its parent t_B.

If a top-level flat- or nested transaction executes inside a subdatabase, then it must commit its WSOCs to that subdatabase, otherwise it should commit its WSOCs to the database.

For example in figure 4.3, Catherine's top-level transaction t_C must commit its WSOC to t_A's subdatabase, while the top-level transactions t_B and t_D must commit to the database.

When a transaction commits to a subdatabase, the owner of the subdatabase will be able to inspect the work done by that transaction. The owner can then decide to delay the inspection, commit the work in order to let other transactions in the subdatabase access the data, or abort the work.

4.3.3 User Sets

An owner of a subdatabase can control who is allowed to access the subdatabase and with which rights. This is done by specifying one or more *user sets* containing user-IDs. Examples of user sets are a user set of readers, a user set of writers, and a user set of readers who can read only with parameter set A. Users in a subdatabase's user set can start transactions inside it and work according to their rights.

4.3.4 Implementation of Subdatabases

(Anfindsen, 1997, page 48) writes the following:

> *Given that concurrency control is carried out by means of locking, subdatabases can be implemented simply by introducing a new lock mode, henceforth referred to as DB.*

A DB lock has the same conflict properties as write locks. So if a transaction holds a DB lock on an item x, then all other locks held on the same item will conflict with the former.

Assume that Alice holds write locks on the objects w, x, y, and z. If Alice wants to put x and y in a subdatabase, then she can convert the actual write locks to a DB lock. This can be done immediately since the write locks and the DB lock have exactly the same conflict relationships to other lock modes.

When x and y are DB locked, they are "contained" in a subdatabase. If Alice specify that Bob and Catherine are allowed to read and write in her subdatabase, then Bob and Catherine can start transactions and aquire read and write locks inside it. If Alice wants to eliminate the subdatabase, then no locks can be held inside it.

As we will se below, some concurrency control scheme must be enforced inside a subdatabase.

4.3.5 An Example of Teamwork in Subdatabases

Recall the teamwork example from section 3.2.1 where Alice, Bob, and Catherine designed the landing gear of an aeroplane together. They needed to modify the same objects before the design could be committed to the outside world.

A solution to how they can carry out their teamwork using subdatabases follows (Anfindsen, 1997, pages 48):

> *When eg Alice reaches a stage in her work when it's necessary to check the electrical interfaces, she calls on Bob to assist her. First she turns her entire work area into a subdatabase, which should be as simple as clicking on a menu option in her design tool window, and filling in 'Bob' when prompted for users that should be given access to this subdatabase. Bob can then lock some of Alice's objects and modify them as appropriate. While Bob is doing this, Alice can work on objects in her subdatabase that he has not locked. When Bob is done, Alice can inspect his work and abort it if necessary (since Bob is committing to Alice and not to the global database). If she's satisfied, she can continue to work on the objects in question, retaining Bob's contributions.*
>
> *Similarly, all landing gear objects for the rear wheels can be contained in a subdatabase to which Alice is given access, so that she can take care of interfacing those objects with the hydraulic system. And if Bob and Catherine need to modify each other's objects, they can create second level subdatabases within the rear wheels subdatabase. Should assistance from a designer outside the landing gear team be needed, the user in question can simply be given access to the pertinent subdatabase.*

4.3.6 Concurrency Control and Correctness in Subdatabases

More than one transaction can execute in a subdatabase at the same time, and it is therefore necessary to enforce some concurrency control scheme in subdatabases.

We will next consider how CSR can be applied recursively in nested subdatabases instead of only in the system level database.

Nested Conflict Serializability (NCSR)

The definition of *nested conflict serializability*, which is the second correctness criterion of Apotram, will now be defined.

Nested Conflict Serializable History. If subdatabases can be nested to arbitrary depths, if transaction histories in subdatabases are CSR, and if transactions in subdatabase histories commit to the subdatabase owner, then the resulting transaction history will be *nested conflict serializable*.

4.3.7 How can NCSR be Enforced?

2PL discussed in section 2.2.3 can be used to enforce CSR in each subdatabase.

Given this and the assumption that a subdatabase is created by converting a set of write locks into a single DB lock, then it can be shown that histories in nested databases will be NCSR (Anfindsen, 1997, page 49). An explanation follows.

A subdatabase which has no further nesting has only serializable histories when 2PL is enforced. If a subdatabase has one or more child subdatabase(s) within it, then all objects in an arbitrary child subdatabase is covered by a DB lock. But since a DB lock behaves like write locks in the parent subdatabase, the parent subdatabse has only serializable histories in it when 2PL is enforced. Then it follows by induction that 2PL in all (sub)databases ensures CSR histories in all (sub)databases which by definition implies NCSR.

4.3.8 NCSR and Recovery

If rigorous 2PL, discussed in section 2.3.4, is used in each subdatabase, then the desired recovery-related properties of nested databases can be ensured. This is shown in (Anfindsen, 1997, page 50). An explanation follows.

If rigorous 2PL is enforced in each subdatabase that has no further nesting, then rigorous histories are ensured in each of them. In subdatabases where one or more child subdatabase(s) exist(s), all objects belonging to an arbitrary child subdatabase are covered by a DB lock. A DB lock has the same behaviour as write locks in the parent subdatabase. If rigorous 2PL is enforced in the parent subdatabase, then that subdatabase will only have rigorous histories. It follows then by induction that rigorous 2PL in all (sub)databases ensures rigorous histories in all (sub)databases. Thus, transaction atomicity and durability is guaranteed. Note that in addition, support for persistent savepoints is desirable

4.3.9 Summing up

It has been shown that write-write conflicts can be handled in a controlled way by letting users convert some or all of their write locks, namely a WSOC, to a DB lock. The DB lock represents a subdatabase where specified users can work. In each subdatabase the concurrency control scheme CSR is ensured. Subdatabases can be nested to an arbitrary depth, and when CSR is enforced in each subdatabase, the resulting correctness criterion is NCSR.

CSR is enforced by 2PL, and in order to get the desired recovery related properties as well, *rigorous* 2PL is enforced in each subdatabase.

4.4 The Correctness Criterion of Apotram

Rather than enforcing CSR in each subdatabase, *CCSR* can be enforced. Then transactions inside a subdatabase can share data by customizing the notion of conflict and give information about uncommitted data to each other.

NCSR and CCSR can easily be *integrated*. With this integration, it is possible to deal with any combination of read-write, write-read, and write-write conflicts.

4.4.1 Nested Conditional Conflict Serializability (NCCSR)

We are now able to define a *nested conditional conflict serializable history*.

Nested conditional conflict serializable history. If subdatabases can be nested to arbitrary depths, if transaction histories in subdatabases are CCSR, and if transactions in subdatabase histories commit to their subdatabase owners, then the resulting transaction history will be *nested conditionally conflict serializable* (NCCSR).

Example

Alice, Bob, and Catherine, as presented in the example in section 4.3.5, worked together on the landing gear of a new aeroplane.

Alice had the responsibility of the front wheels while Bob and Catherine had the responsibility of the rear wheels. In addition Bob had the responsibility of interfacing all objects with the electrical system, and Alice had the responsibility of interfacing all objects with the hydraulic system.

Recall that Alice put her front wheel objects in a subdatabase in order to let Bob do the electrical interfacing. While Bob is working on these objects he can choose to use a write parameter set if he is willing to share his data with other transactions.

Then other transactions, e.g. transactions that need to look at the electrical interfacing in order to do their designs, can use read parameter sets in order to make their read modes compatible with Bob's parameterized write mode. Note that it is necessarily assumed that the users of these transactions have rights to read access Alice's subdatabase.

Parameterized DB lock

If a transaction shall create a subdatabase and if its objects are write locked in a parameterized mode, then the subdatabase can be parameterized too. It is not obvious how this should be done, but that and other parameter issues will be discussed in chapter 5 and chapter 6.

4.4.2 Summing up

With the correctness criterion NCCSR we have seen that any combination of read-write, write-read, and write-write conflict can be dealt with. Transactions can create subdatabases to an arbitrary depth and CCSR is the correctness criterion enforced in each subdatabase as well as in the system level database.

4.5 Summary

Apotram is developed in order to meet the cooperation requirements of advanced transactions.

The correctness criterion CCSR is an extension of CSR and it enables transactions to share data in a controlled fashion. Transactions can have a read parameter set and a write parameter set which influence the notion of conflict.

We saw how CCSR could be enforced by following the 2PPL protocol and how the A and D of ACID could be achieved.

The correctness criterion NCSR is also an extension of CSR and it enable transactions to do teamwork in a controlled manner. A transaction can convert its write locks to a DB lock, resulting in a subdatabase creation. Users can be allowed to access the subdatabase and CSR can be enforced inside each subdatabase in order to achieve concurrency control.

We saw how CSR could be enforced in each subdatabase by following the 2PL protocol, and how the A and D of ACID could be achieved.

Finally we saw how the two correctness criteria could be combined to the correctness criterion NCCSR of Apotram just by ensuring CCSR in each subdatabase instead of CSR.

Chapter 5

Parameterized DB locks

We will in this chapter consider issues concerning parameterized DB locks. Recall first that a subdatabase is created by converting one or more write locks into one single DB lock. Given that parameters are used and that CCSR is the correctness criterion enforced in each subdatabase, there are some open issues to discuss. (Anfindsen, 1997, page 51) writes the following:

> *Three questions must be answered:*
>
> 1. *If the write locks that are to be converted to a DB lock have different parameters, what should be the parameter set of the resulting DB lock?*
> 2. *If the parameters of a DB lock and the write locks within its domain differ, what parameters should readers that encounter the DB lock see?*
> 3. *Should there be restrictions on the write parameters that may be used within the domain of a subdatabase?*

The first question will be discussed in section 5.1, and the two next in section 5.2.

5.1 Write to DB Lock Conversion

In general a transaction can have objects write locked in different parameterized modes. Then if these objects are to be put in a subdatabase, there is a question which parameter set should be attached to the DB lock (Anfindsen, 1997, page 51). But in this thesis it will be assumed that a transaction only can have *one* wps and *one* rps. Then a DB lock can naturally have the same wps as the transaction who is the subdatabase creator.

We will now take a look at the assumptions made. We will assume that nested transactions are supported. (Anfindsen, 1997, pages 53–60) argues that nested transactions are important in order to support certain features.

He also argues that write mode parameters should be specified at the level of subtransactions instead of e.g. data items (Anfindsen, 1997, page 55). It will in this thesis therefore be assumed that each transaction in a nested transaction can have *one* wps and *one* rps. If there is no nesting, then a flat transaction can similarly keep *one* wps and *one* rps. With this assumption, all write locked objects of a transaction will be locked in the same mode, and all read locked objects will be locked in the same mode.

While it here is assumed that a subdatabase has the same wps as its creator, (Anfindsen, 1997, page 51) discusses some possible solutions to this question in general, like to require that all write locks which is to be converted to a DB lock must have equal wpss before the transformation can be executed.

5.2 Relationship of DB Locks to Read Locks

5.2.1 Subdatabase- and foreign transactions

First the concepts of subdatabase- and foreign transactions will be introduced.

While a subdatabase exists it will have user sets associated with it, as we saw in section 4.3.3. E.g. some users will be allowed to spawn transactions that can write and read objects in it, and some other users will be allowed to spawn transactions that can only read objects in it. Transactions *begun in the subdatabase*, consequently owned by users belonging to one of the subdatabase's user sets, will be called *subdatabase transactions*, *subdatabase readers*, or *subdatabase writers* depending on the context.

Transactions *begun outside the subdatabase* may need to read data in the subdatabase too, like transactions that held parameterized read locks on one or more items that were write locked by the subdatabase creator when the subdatabase was created. Let us call these transactions *foreign transactions*, *foreign readers*, or simply *foreigners* (to a certain subdatabase).

It is assumed that foreign transactions of a certain subdatabase are allowed to read objects in it even though their owners may not be members of any of the subdatabase's user sets. However, foreign transactions may not modify objects in a subdatabase.

5.2.2 What wps(s) shall a reader see?

(Anfindsen, 1997, page 52) writes the following:

> *An r(A) lock is compatible with w(B) and DB(B) locks if and only if $B \subseteq A$. When an r(A) lock is placed on a data item covered by a w(B) lock, the reader sees the parameter set B. But what parameter set should a reader see when an r(A) lock is placed on a data item covered by a DB(B) lock?*

This is a problem since an object in a subdatabase may be locked by a subdatabase transaction in a parameterized write mode, and the wps of the subdatabase transaction may in general be different from the wps B of the subdatabase. Further nesting of subdatabases may also occur. So theoretically an object can be covered by arbitrarily many DB locks and one write lock at the *same* time, where some or all write modes may be different. Which wps(s) should readers see? For an example we will in a moment consider figure 5.1.

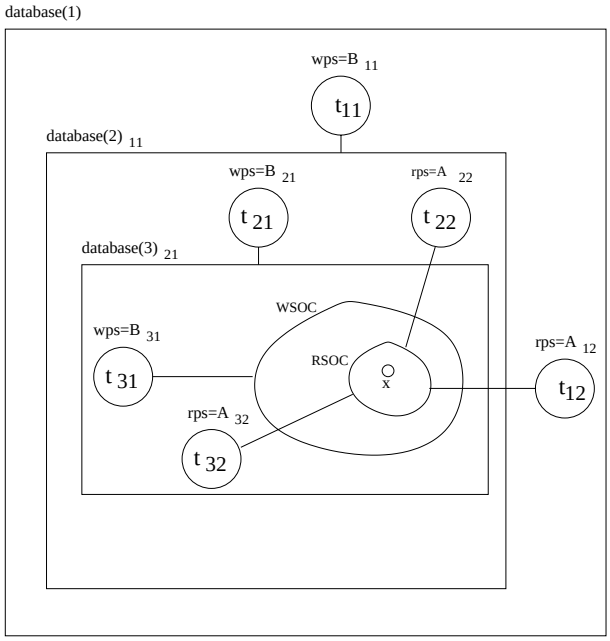

Figure 5.1: The object x has two DB locks and one write lock. Then three, possibly different, wpss are relevant to x.

Notation

Before we consider figure 5.1, a comment about the notation in the figure will be given.

All transactions are denoted with the letter t. The subscript of t consists of two digits. The first digit indicates the database level that the transaction is being executed in. E.g. a transaction being executed in the top-level database has a subscript that starts with the digit *1* because it executes in the database level 1. The second digit indicates the *transaction number* in the actual database, causing e.g. the second transaction started in the top-level database to have the subscript *12*. The top-level database is denoted *database*(1). A subdatabase is denoted by the word *database* followed by

its level in parenthesis, and with two digits as a subscript indicating which transaction has created it. As an example $database(3)_{21}$ has level 3 and is created by transaction t_{21}.

Example

We will now consider figure 5.1. The object x is write locked by t_{31}, while t_{21} and t_{11} hold DB locks on it. This means that three wpss are relevant to the object x. In addition t_{12}, t_{22}, and t_{32} hold parameterized read locks on x.

Transaction t_{12} is a foreigner to both subdatabases. Which wps(s) should it see when it reads x? The transaction t_{22} is a subdatabase transaction in $database(2)_{11}$ but a foreigner to $database(3)_{21}$. Which wps(s) should this transaction see when it reads the object x? And finally which wps(s) should transaction t_{32}, that is a subdatabase transaction in $database(3)_{21}$, see when it reads x?

Possible answers to these questions will be discussed later in this section.

More Terminology

Before the discussion begins, it is necessary to introduce some terminology.

Database tree. A top-level database and all subdatabases nested in it can be mapped to a *database tree*. The root node in the tree represents the top-level database, and all other nodes represent subdatabases. If a (sub)database has no further subdatabases in it, then its subdatabase node will be called a leaf. If the top-level database has no subdatabases in it, then the tree has only one node.

For an example we will in a moment consider figure 5.2, that shows a certain top-level database and its subdatabase nesting.

Comment. Before we see what the nested (sub)databases' corresponding tree looks like, a comment about the database and transaction names used will be made.

In figure 5.2 there are two transactions with the same name, and also two subdatabases with the same name. However, there are no two names that are alike in the *same* (sub)database. It is not a problem for the discussions that follow if transactions or subdatabases have the same name in different databases. The notation is therefore chosen because of its simplicity.

Example of a Database Tree

The corresponding database tree of the (sub)databases in figure 5.2 is shown in figure 5.3 where $(i)_{(i-1)j}$ is a shorthand notation for $database(i)_{(i-1)j}$.

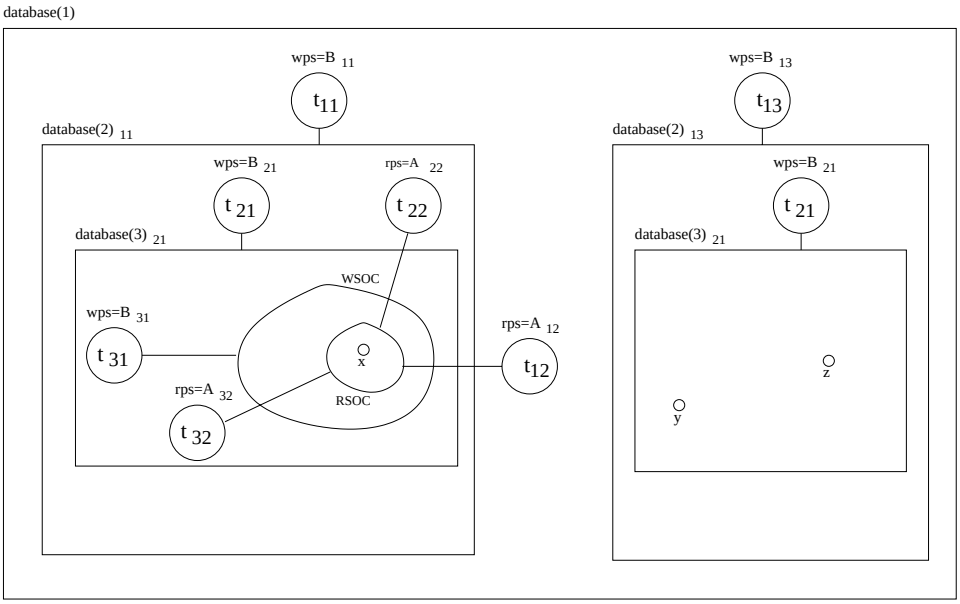

Figure 5.2: Nested (sub)databases.

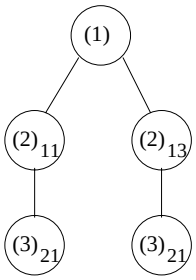

Figure 5.3: The nested (sub)databases' corresponding database tree.

Descendants, Inferiors, Ancestors, and Superiors

We will need the concepts of a certain node's *descendants*, *inferiors*, *ancestors*, and *superiors*. Descendants of a certain node in a tree are the node itself and all nodes in the subtree where the node in question is the root. E.g. in figure 5.3, the descendants of node $(2)_{11}$ are $(2)_{11}$ and $(3)_{21}$. Inferiors of a certain node are the descendants except the node itself. E.g. in figure 5.3, the inferior of $(2)_{11}$ is $(3)_{21}$. Ancestors of a certain node are the node itself and all nodes in the tree from that node on the straight line to the root node. E.g. in figure 5.3, the ancestors of $(2)_{11}$ are $(2)_{11}$ and (1). Superiors of a certain node are the ancestors except the node itself. E.g. in figure 5.3, the ancestors of $(2)_{11}$ is only (1).

Database Tree Relevant to a Certain Object

It will be interesting to talk about the *database tree relevant to an arbitrary object x*. That tree has a node for each database that contains x. Note that no node in a tree relevant to a certain object can have siblings because no two transactions executing in the same database can have DB locks on a given object at the *same* time.

The resulting database tree relevant to x corresponding to the nested (sub)databases in figure 5.2 is shown in figure 5.4.

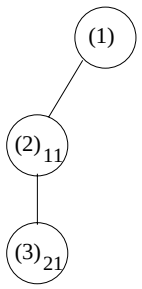

Figure 5.4: The nested (sub)databases' corresponding database tree relevant to x.

Discussion of the Initial Question

We will next discuss three possible solutions to the problem of which wps(s) a transaction should see for a given object that has more than one relevant wps associated with it.

It is assumed in the rest of this section that all transactions executing in subdatabases have parameter sets and that all transactions holding a DB lock in the top-level database have wpss. Transactions executing in the top-

level database that only hold read locks and/or write locks may choose not to have parameter sets if that is possible.

It is also assumed that the parameter sets are static. Then transactions cannot change their parameter sets. Dynamic modification of parameter sets will first be discussed in chapter 6.

According to the notation presented above, the top-level database should be denoted as $database(1)$. But for simplicity, please let $database(i)_{(i-1)j}$ be interpreted as $database(1)$ if i = 1 in the following.

5.2.3 Solution 1

Assume that there is a transaction t_{ir}, executing in $database(i)_{(i-1)j}$, that holds a read lock on the object x. Solution 1 requires that all possibly existing wpss which are relevant to x shall be visible for t_{ir}. Then if x is contained in one or more subdatabases in the database tree, each of these subdatabases' corresponding DB locks have to be compatible with t_{ir}'s read lock. Also if x is write locked by some transaction in the leaf database of the database tree relevant to x, then that write lock and t_{ir}'s read lock have to be compatible.

Example

As an example we will now consider figure 5.1. The object x has, as we saw above, three wpss that are relevant to it, namely the ones of t_{11} (or $database(2)_{11}$), t_{21} (or $database(3)_{21}$), and t_{31}. According to solution 1, all three readers, namely t_{12}, t_{22}, and t_{32}, will see all three wpss. Then each of these reading transactions' rpss must be equal to or supersets of all the wpss that are relevant to x, and consequently A_{12}, A_{22}, and A_{32} must be equal to or supersets of the union of B_{11}, B_{21}, and B_{31}.

The Algorithms

The algorithms used to decide whether a certain lock can be given or not will now be presented. Note that the goal is to give the logical idea, and not any programming details.

Can a certain read lock be given? Assume that t_{ir}, executing in $database(i)_{(i-1)j}$, aquires a *read* lock on the object x. The system must do as shown in figure 5.5 in order to decide whether the lock can be given or not. The subalgorithms used by the algorithm shown in figure 5.5 are presented in figure 5.6, and figure 5.7.

The explanation that follows is very detailed, so the experienced reader may skip to page 56.

Consider figure 5.5. The algorithm checks if i = 1 in order to find out if $database(i)_{(i-1)j}$ has any superior databases. If i = 1, then t_{ir} is executing

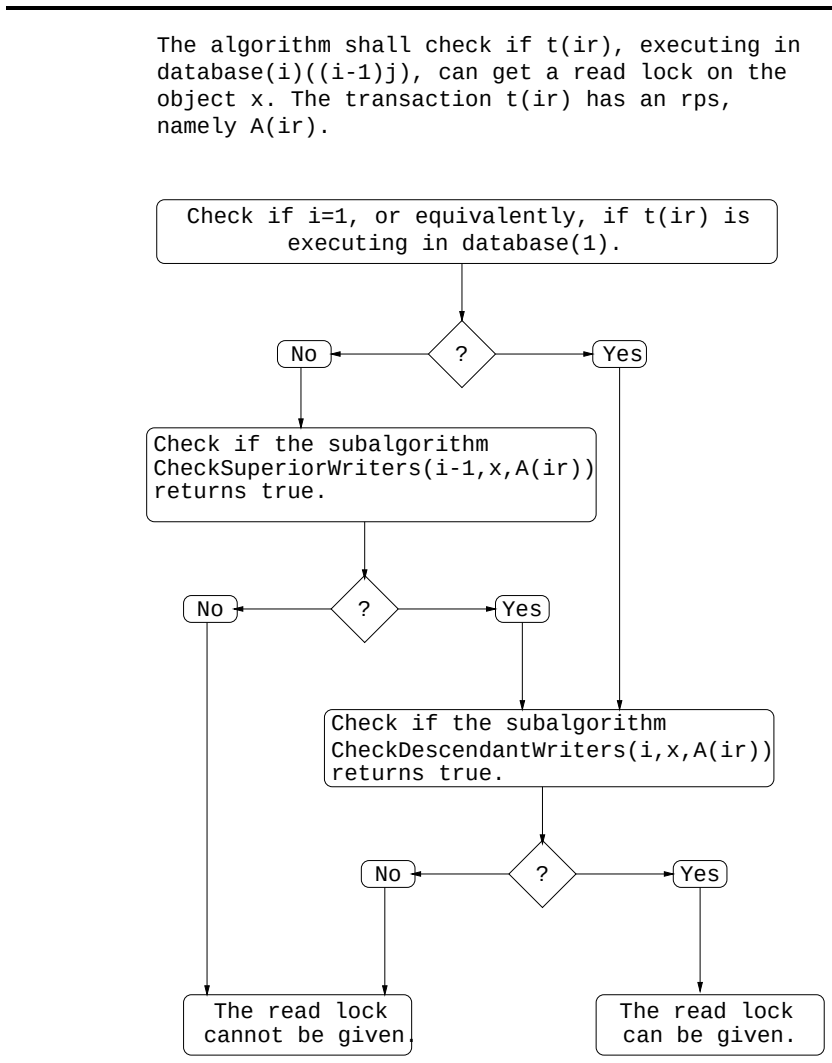

Figure 5.5: The algorithm used to decide if a transaction can get a read lock in solution 1.

The algorithm shall check if t(ir)'s desired read lock on the object x in database(i)((i-1)j), is compatible with the DB lock(s)held on x in database(i)((i-1)j)'s superior database(s). The algorithm has three input parameters. The first one is the integer k, indicating which database level the checking should start on, the second is the object x, or some identification of it, to specify which database on level k that is to be checked, and the third is t(ir)'s rps, namely A(ir), that will be used to decide whether the DB locks on x in database(i)((i-1)j)'s superior databases are compatible with t(ir)'s desired read lock.
Note that more than one database can exist on a given database level, but that only one of these databases can contain a certain object, since no two transactions in the same database can DB- or write lock a certain object simultaneously. Therefore the two first parameters tell exactly which database to check.

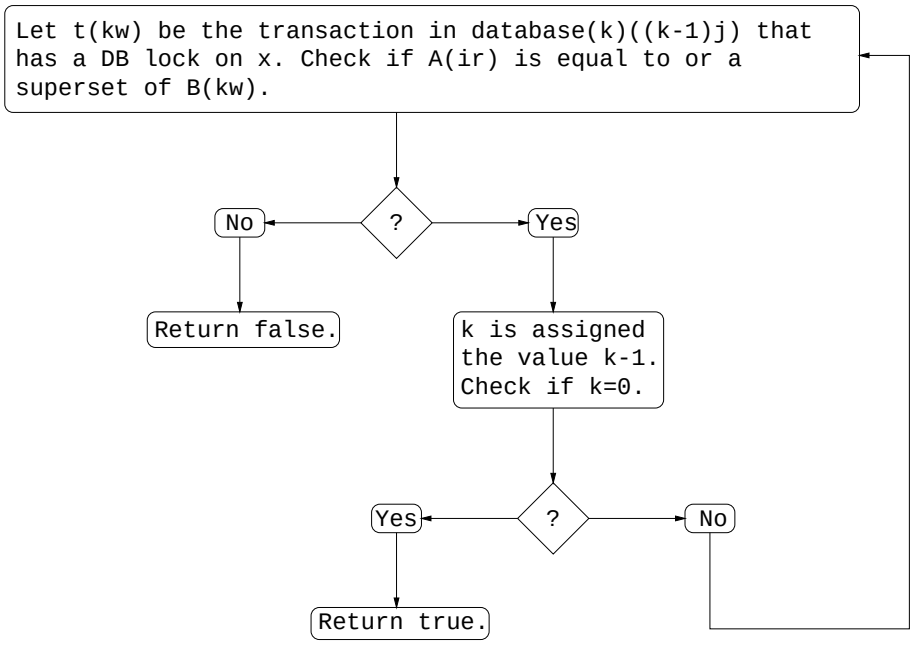

Figure 5.6: The subalgorithm CheckSuperiorWriters(k,x,A_{ir}).

```
The algorithm shall check if t(ir)'s desired read lock on
the object x in database(i)((i-1)j) is compatible with the
possibly held DB- and write locks on x in the descendant
database(s) of database(i)((i-1)j). The algorithm has three
input parameters. The first one is the integer k indicating
which database level the checking should start on, the
second is the object x, or some identification of it, to
specify which database on level k that is to be checked,
and the third is t(ir)'s rps, namely A(ir) that will be used
to decide whether the possibly held  DB- or write lock on
x in this descendant database of database(i)((i-1)j)
is compatible with t(ir)'s desired read lock.
```

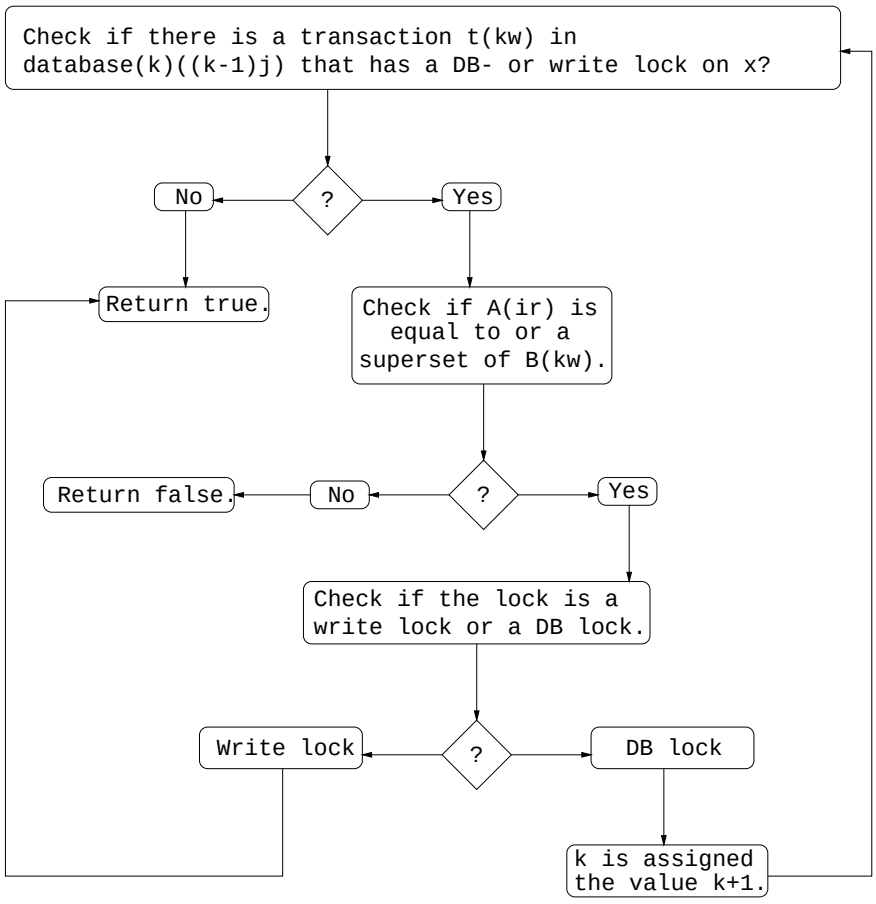

Figure 5.7: The subalgorithm CheckDescendantWriters(k,x,A_{ir}).

in $database(1)$, and no superior databases exist. But if i > 1, then $database(i)_{(i-1)j}$ has superior database(s), and they must be checked in order to find out whether t_{ir}'s desired read lock on the object x is compatible with the DB lock(s) held on the object x in the superior database(s) of $database(i)_{(i-1)j}$.

The subalgorithm *CheckSuperiorWriters*, shown in figure 5.6, checks this recursively. It starts to check $database(i)_{(i-1)j}$'s parent, as indicated by the first and second parameters. It ends with the top-level database or with the first database it finds containing a DB lock on x which is incompatible with t_{ir}'s desired read lock. If at least one superior database of $database(i)_{(i-1)j}$ contains a DB lock on x which is incompatible with t_{ir}'s desired read lock, then the subalgorithm returns false. Otherwise it returns true.

If the subalgorithm CheckSuperiorWriters returns true or if t_{ir} is executing in $database(1)$, then it must be checked if there are held any DB- or write lock(s) on x in the descendant database(s) of $database(i)_{(i-1)j}$ that are incompatible with t_{ir}'s desired read lock. The subalgorithm *CheckDescendantWriters*, shown in figure 5.7, checks this recursively. Note that it is only necessary to check the descendant database(s) of $database(i)_{(i-1)j}$ which *also* exist in the database tree *relevant to x*.

CheckDescendantWriters starts to check $database(i)_{(i-1)j}$, as indicated by the first and second parameters. It ends with the leaf database of the database tree relevant to x or with the first database it finds containing a DB- or write lock that is incompatible with t_{ir}'s desired read lock. If it is held at least one DB- or write lock on x in $database(i)_{(i-1)j}$'s descendant databases that is incompatible with t_{ir}'s desired read lock, then the subalgorithm returns false. Otherwise it returns true.

If one of the subalgorithms returns false, then it means that there exists at least one transaction holding a DB- or write lock on x which is incompatible with t_{ir}'s desired read lock. Then t_{ir}'s desired read lock on x cannot be given.

However, if t_{ir} is executing in $database(1)$ and the subalgorithm CheckDescendantWriters returns true or if t_{ir} is executing in $database(i)_{(i-1)j}$ where i > 1 and both subalgorithms return true, then it means that no DB- or write lock held on x is incompatible with t_{ir}'s desired read lock. Consequently, t_{ir}'s desired read lock on x can be given.

A detailed explanation of the subalgorithm CheckSuperiorWriters will be given next. Consider figure 5.6. The algorithm checks if t_{ir}'s rps is equal to or a superset of t_{kw}'s wps. If it is not, then t_{ir}'s desired read lock is incompatible with t_{kw}'s DB lock. Then the subalgorithm has found that at least one DB lock on x in $database(i)_{(i-1)j}$'s superior database(s) is incompatible with t_{ir}'s desired read lock, and it returns false.

But if $A_{ir} \supseteq B_{kw}$, then t_{ir}'s desired read lock is compatible with t_{kw}'s DB lock on x. Then CheckSuperiorWriters has found that $database(k)_{(k-1)j}$ is ok, and if there is one, the next superior database of $database(i)_{(i-1)j}$ can

be checked. In order to find out if there is another superior database of $database(i)_{(i-1)j}$, k is reduced by one. If k $\geq$ 1 then there is one or more superior databases left to check, and the subalgorithm is repeated with the new value of k as an input parameter together with x and A_{ir}, as before. Otherwise, if k = 0, then all superior databases of $database(i)_{(i-1)j}$ have been checked. The algorithm has then found that all DB locks held on x in $database(i)_{(i-1)j}$'s superior databases are compatible with t_{ir}'s desired read lock, and it returns true.

A detailed explanation of the subalgorithm CheckDescendantWriters will be given next. Consider figure 5.7. The algorithm checks if there is a transaction t_{kw}, in $database(k)_{(k-1)j}$ that has DB- or write locked x.

If there is, then it is checked if t_{ir}'s rps A_{ir} is equal to or a superset of t_{kw}'s wps B_{kw}. If it is not, then t_{ir}'s desired read lock is incompatible with t_{kw}'s DB- or write lock. Then the subalgorithm has found at least one DB- or write lock on x in the descendant database(s) of $database(i)_{(i-1)j}$ that is incompatible with t_{ir}'s desired read lock, and it returns false. However, if $A_{ir} \supseteq B_{kw}$, then t_{ir}'s desired read lock is compatible with t_{kw}'s DB- or write lock. If the lock is a DB lock, then the next descendant database of $database(i)_{(i-1)j}$ must be checked. In order to do so, one is added to k, and the subalgorithm is repeated with the new value of k as an input parameter together with x and A_{ir}, as before. But if the lock is a write lock, then we know that the leaf database of the database tree relevant to x is reached, and the subalgorithm has found that all possibly held DB- and write lock(s) on x in the descendant database(s) of $database(i)_{(i-1)j}$ are compatible with t_{ir}'s desired read lock. Then the subalgorithm returns true.

Also if no transaction has DB- or write locked x in $database(k)_{(k-1)j}$, then we know that $database(k)_{(k-1)j}$ is the leaf database in the database tree relevant to x. This means that no more DB- or write locks are held on x in the descendant databases of $database(k)_{(k-1)j}$. The subalgorithm has then found that there exists no DB- or write locks in the descendant database(s) of $database(i)_{(i-1)j}$ that are incompatible with t_{ir}'s desired read lock, and it returns true.

Can a certain write lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *write* lock on the object x. The system must do as shown in figure 5.8 in order to decide whether the lock can be given or not. The subalgorithm used by the algorithm shown in figure 5.8 is presented in figure 5.9.

The explanation that follows is very detailed, so the experienced reader may skip to page 60.

Consider figure 5.8. The algorithm checks if there are another transaction than t_{iw} in $database(i)_{(i-1)j}$ that holds a DB- or write lock on the object x.

If there is, then the write lock cannot be given. However, if no other

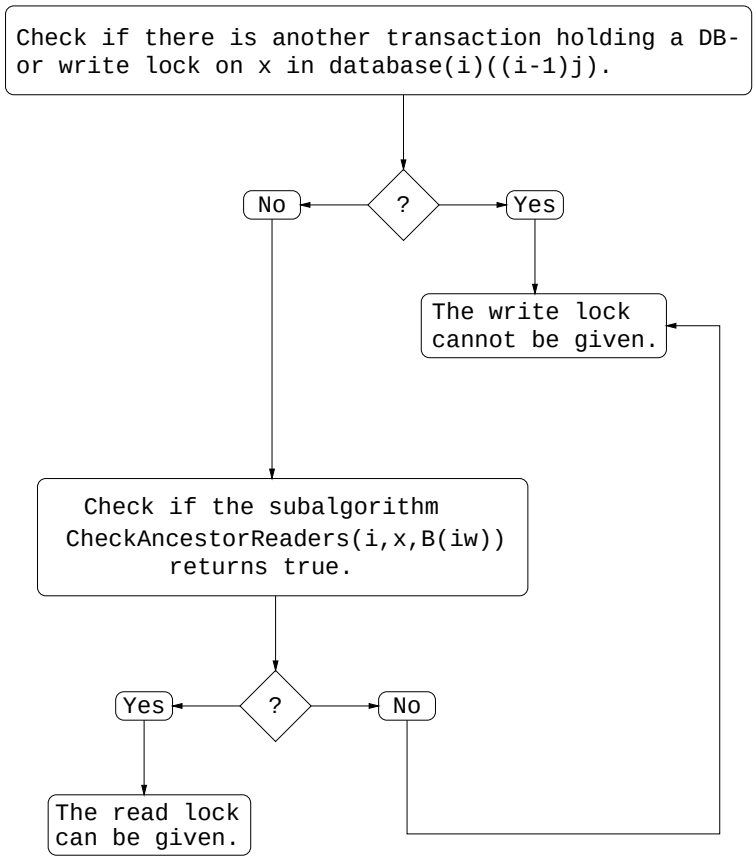

Figure 5.8: The algorithm used to decide if a transaction can get a write lock in solution 1 and 2.

The algorithm shall check if t(iw)'s desired write lock
on the object x in database(i)((i-1)j), is compatible
with possibly existing read locks held on x in
the ancestor databases of database(i)((i-1)j).
The algorithm has three input parameters.
The first one is the integer k, indicating which
database level the checking should start on, the
second is the object x, or some identification of it, to
specify which database on level k that is to be checked,
and the third is t(iw)'s wps, namely B(iw), that will be
used to decide whether the possibly existing read locks
on x in the ancestor databases of database(i)((i-1)j),
are compatible with t(iw)'s desired write lock.

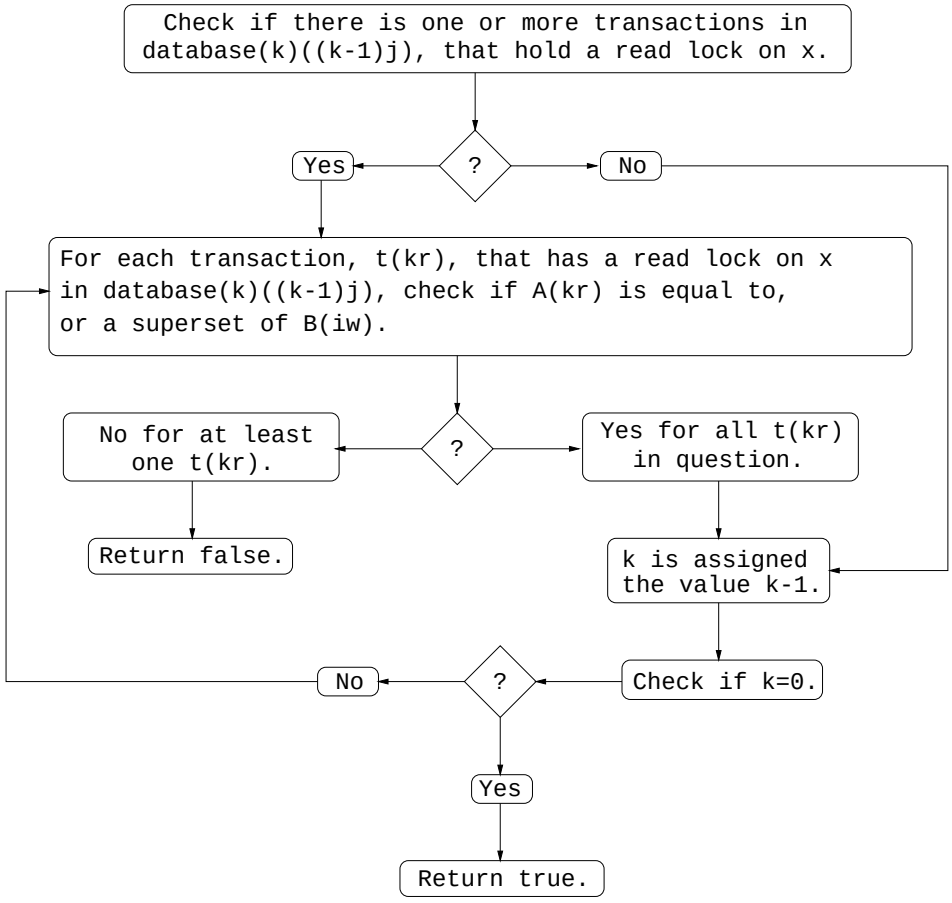

Figure 5.9: The subalgorithm CheckAncestorReaders(k,x,B_{iw}).

transaction in $database(i)_{(i-1)j}$ holds a DB- or write lock on x, then the next step is to check if all possibly existing transactions holding read locks on x, have read locks that are compatible with t_{iw}'s desired write lock.

Note that when no transaction holds a DB lock on x in $database(i)_{(i-1)j}$, we know that this database is the leaf database in the database tree relevant to x, and consequently this database has only one descendant database, namely $database(i)_{(i-1)j}$ itself. Therefore, possibly existing transactions holding read locks on x must be executing in the ancestor database(s) of $database(i)_{(i-1)j}$.

The subalgorithm CheckAncestorReaders checks recursively if all possibly existing transactions that hold read locks on x have read locks that are compatible with t_{iw}'s desired write lock. It starts to check the transactions in $database(i)_{(i-1)j}$, as indicated by the two first parameters of the subalgorithm. It ends with the top-level database or the first database found with a transaction holding a read lock that is incompatible with t_{iw}'s desired write lock. If CheckAncestorReaders actually finds a transaction that holds a read lock on x which is incompatible with t_{iw}'s desired write lock, then it returns false. Otherwise, if it does not find a such transaction, then it returns true, and consequently t_{iw}'s desired write lock can be given.

The subalgorithm CheckAncestorReaders will now be explained in detail. Consider figure 5.9. The subalgorithm checks if there are any transactions in $database(k)_{(k-1)j}$ that hold a read lock on x. If there are, then each of them is checked to find out if their rpss are equal to or supersets of t_{iw}'s wps. If at least one transaction holds a read lock that is incompatible with t_{iw}'s desired write lock, then the subalgorithm has found that at least one transaction in the ancestor databases of $database(i)_{(i-1)j}$ hold a read lock that is incompatible with t_{iw}'s desired write lock. The subalgorithm then returns false. However, if all transactions in $database(k)_{(k-1)j}$ that hold a read lock on x have rpss that are equal to or supersets of t_{iw}'s wps, then the subalgorithm concludes that $database(k)_{(k-1)j}$ is ok. If there is one, then the next ancestor database of $database(i)_{(i-1)j}$ must be checked. In order to find out if there is another ancestor, k is reduced by one. If $k > 0$ then there is one or more ancestor database(s) left to check, and the subalgorithm is repeated with the new value of k as an input parameter together with x and B_{iw}, as before. But if $k = 0$, then there are no more ancestor databases to check, and the subalgorithm has found that there are no transactions holding a read lock on x that is incompatible with t_{iw}'s desired write lock. The subalgorithm then returns true.

If there are no transactions in $database(k)_{(k-1)j}$ that hold read locks on the object x, then the subalgorithm concludes that $database(k)_{(k-1)j}$ is ok. Then if there is one, the next ancestor database of $database(i)_{(i-1)j}$ must be checked. As above, in order to find out if there is another ancestor, k is reduced by one. If $k > 0$, then there is one or more ancestor database(s) left to check, and the subalgorithm is repeated with the new value of k as

an input parameter together with x and B_{iw}. But if k = 0, then there are no more ancestor databases to check, and the subalgorithm has found that there are no transactions holding a read lock which is incompatible with t_{iw}'s desired write lock. The subalgorithm then returns true.

Can a certain DB lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *DB* lock on the object x. If it is assumed that t_{iw} already has a write lock on the object x, then this lock can be given right away since the compatibility properties of write locks and DB locks are the same.

Discussion of Solution 1 and Introduction of Parameter Usage Rules

A discussion of solution 1 follows. In this solution we have seen that it is required that *all* locks held on a certain object in the database tree must be compatible.

Then the arbitrary transaction t_{ir}, holding a read lock on the object x in $database(i)_{(i-1)j}$, must have an rps which is the union of *all* wpss relevant to the object x in the database tree. This means that a transaction holding a read lock on a certain object will get all information given in wpss belonging to transactions holding DB- or write locks on that object.

In order to find out whether a certain lock can be given or not in this solution, a lot of checking may be the result if the database tree relevant to the actual object is deep. This is because it is necessary to check if there are held conflicting locks on the actual object in all databases where the object is contained.

All the checking may therefore make this solution quite complicated. But if it *is* desirable to let all readers of an object see all wpss relevant to it, then restrictions on parameter usage in subdatabases can be made in order to get a less complicated checking process when the system shall decide whether a certain lock can be given or not.

The rules presented in figure 5.10 restrict the parameter usage in subdatabases. As we will see below, when these rules are enforced it will be easier to decide whether a certain lock can be given or not.

The new Algorithms

The new algorithms used to decide whether a certain lock can be given or not will now be presented.

Can a certain read lock be given? Assume that t_{ir}, executing in $database(i)_{(i-1)j}$, aquires a *read* lock on the object x. The system must do as shown in figure 5.11 in order to decide whether the lock can be given or not.

Let the subdatabase transactions t(iw) and t(ir) execute in database(i)((i-1)j) where i>1, and let t(1j) be the transaction holding the DB lock corresponding to the subdatabase which is database(i)((i-1)j)'s superior in database(1). The rules are as follows:

1. The wps of t(iw) must be equal to or a subset of database(i)((i-1)j)'s wps B((i-1)j).
2. The rps of t(ir) must be equal to or a superset of t(1j)'s wps B(1j).

Figure 5.10: The rules that restrict the subdatabase parameter usage in solution 1.

The algorithm shall check if t(ir), executing in database(i)((i-1)j), can get a read lock on the object x. The transaction t(ir) has an rps, namely A(ir).

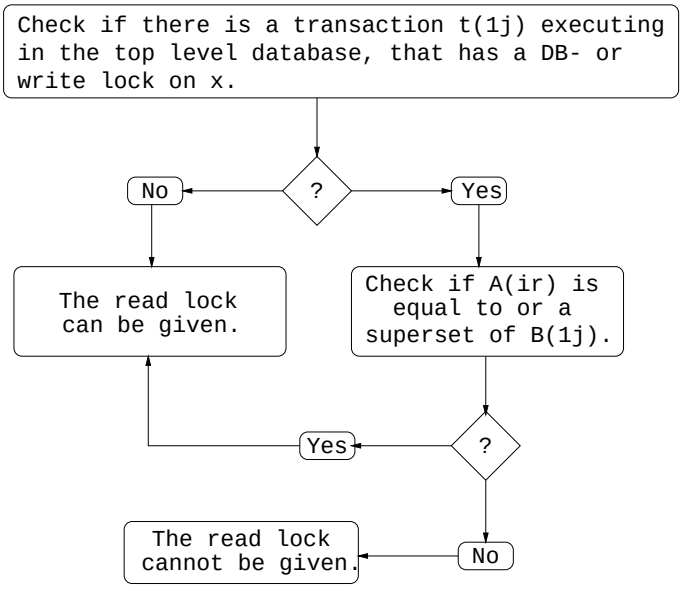

Figure 5.11: The algorithm used to decide if a transaction can get a read lock in solution 1, when the parameter usage rules for subdatabases shown in figure 5.10 are enforced.

A detailed explanation of the algorithm follows. Consider figure 5.11. The algorithm checks if there is a transaction t_{1j} in the top-level database that has a DB- or write lock on x. If it is not, then we know that x is contained only in $database(1)$, and that t_{ir} is executing in that database. Then since no transaction holds a lock that conflicts with t_{ir}'s desired read lock, the lock can be given. However, if x is DB- or write locked by t_{1j}, then it must be checked if $A_{ir} \supseteq B_{1j}$. This is the only wps that has to be checked because all other possibly existing wpss relevant to x will be equal to or subsets of B_{1j}, according to rule 1 shown in figure 5.10. So if $A_{ir} \supseteq B_{1j}$, then we know that A_{ir} is equal to or a superset of all wpss relevant to x, and t_{ir}'s desired read lock can be given. Otherwise, it cannot be given.

Can a certain write lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *write* lock on the object x. The system must do as shown in figure 5.12 in order to decide whether the lock can be given or not.

A detailed explanation of the algorithm follows. Consider figure 5.12. The algorithm checks first if there is another transaction in $database(i)_{(i-1)j}$ that holds a DB- or write lock on the object x. If there is, then the write lock cannot be given. If there is not, then it must be checked whether t_{iw} is executing in $database(1)$ or not, or equivalently if i = 1.

This is because if t_{iw} is executing in $database(1)$, then possibly existing read locks held on x in $database(1)$ must be checked in order to find out if they are compatible with t_{iw}'s desired write lock. If i > 1 then, t_{iw} is not executing in $database(1)$, and it will not be necessary to check the possibly existing read locks on x. This is because each transaction holding a read lock on x must have an rps that is equal to or a superset of the wps of t_{1j} (according to rule 2 shown in figure 5.10) and because t_{iw}'s wps is equal to or a subset of t_{1j}'s wps (according to rule 1 shown in figure 5.10).

So if i > 1, then the write lock can be given right away.

If i = 1, then each possibly held read lock on x in $database(1)$ must be checked in order to find out whether it is compatible with t_{iw}'s desired write lock or not. If all are, then the lock can be given. Otherwise, if at least one read lock held on x in $database(1)$ is incompatible with t_{iw}'s desired write lock, then it cannot be given.

The Consequences of Enforcing the Parameter Usage Rules

The consequences of enforcing the rules presented in figure 5.10 will now be discussed.

Recall that solution 1 is interesting if it is desirable that all transactions holding a read lock on a certain object shall see *all* wpss relevant to that object. In order to get less complicated algorithms for deciding whether

The algorithm shall check if t(iw), executing in database(i)((i-1)j), can get a write lock on the object x. The transaction t(iw) has a wps, namely B(iw).

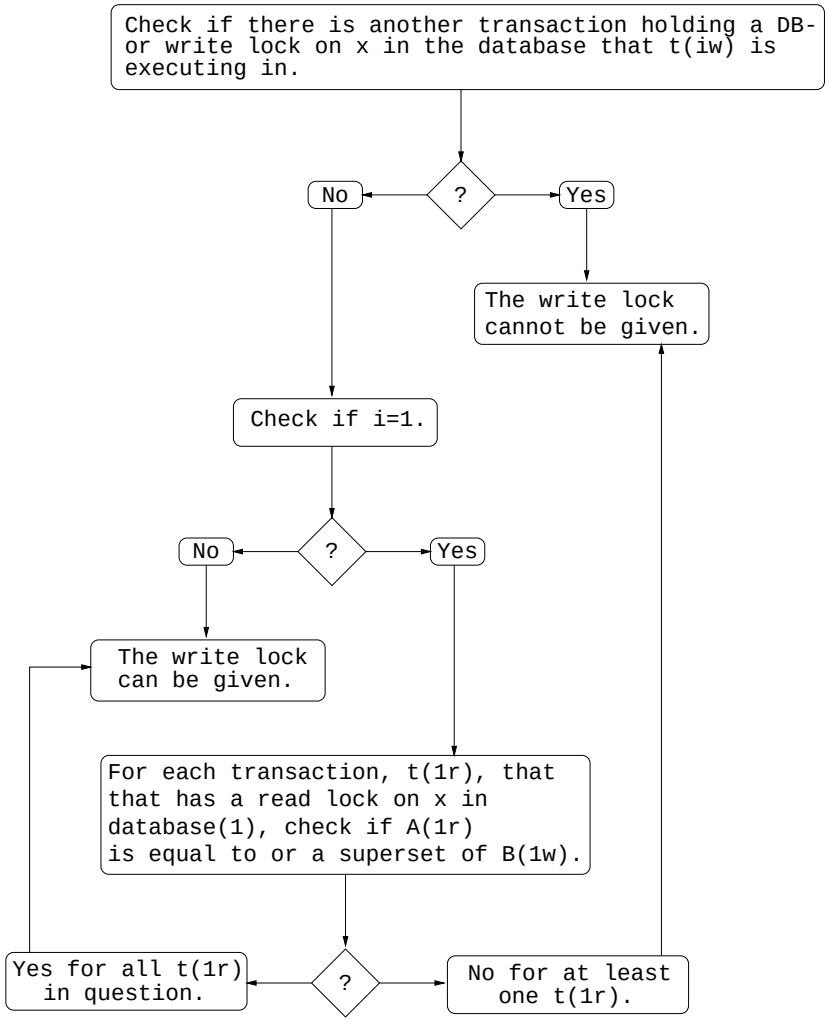

Figure 5.12: The algorithm used to decide if a transaction can get a write lock in solution 1, when the parameter usage rules for subdatabases shown in figure 5.10 are enforced.

a certain lock should be given or not, the rules shown in figure 5.10 were presented.

By enforcing these parameter set usage rules, the algorithms in solution 1 become easier. Instead of checking all wpss relevant to the object x when the transaction t_{ir} aquires a read lock on it, it is sufficient to check only the wps of the possibly existing transaction t_{1j}, holding a DB- or write lock on the actual object in $database(1)$.

When t_{iw} aquires a write lock and it is found that no other transaction in $database(i)_{(i-1)j}$ holds a DB- or write lock on x, then instead of checking all possibly existing read locks held on the actual object in the database tree, it is sufficient to either check the possibly existing readers in $database(1)$ if the writer is executing in that database, or none at all if the writer executes in some inferior database of $database(1)$.

A DB lock can as before be given right away, assumed that write locks are held initially to the DB lock request on the actual object(s).

While the algorithms become easier, less freedom is gained for subdatabase transactions in their parameter set usage. A subdatabase writer t_{iw}, executing in $database(i)_{(i-1)j}$, can only use a wps that is equal to or a subset of the wps of $database(i)_{(i-1)j}$, and a subdatabase reader t_{ir}, executing in $database(i)_{(i-1)j}$, must have an rps that is equal to or a superset of the wps of the transaction t_{1j}, holding a DB lock on the actual object in the relevant superior database of $database(i)_{(i-1)j}$ in $database(1)$.

5.2.4 Solution 2

Assume that there is a transaction t_{ir}, executing in $database(i)_{(i-1)j}$, that holds a read lock on the object x. Solution 2 requires that all possibly existing wpss which are relevant to x in the *descendant* database(s) of $database(i)_{(i-1)j}$, shall be visible for t_{ir}. Then the possibly existing DB locks on x in these databases have to be compatible with t_{ir}'s read lock. In addition, if x is write locked by some transaction in the leaf database of the database tree relevant to x, then that write lock and t_{ir}'s read lock have to be compatible.

Example

As an example, we will again consider figure 5.1. The transaction t_{12} is executing in $database(1)$. The wpss relevant to x in the descendant databases of $database(1)$ are B_{11}, B_{21}, and B_{31}. So t_{12} will see these three wpss. Then A_{12} must be equal to or a superset of the union of B_{11}, B_{21}, and B_{31}.

Let us consider t_{22} executing in $database(2)_{11}$ next. The wpss relevant to x in the descendant databases of $database(2)_{11}$ are B_{21}, and B_{31}. So t_{22} will see these two wpss. Then A_{22} must be equal to or a superset of the union of B_{21} and B_{31}.

Finally, consider t_{32}, executing in $database(3)_{21}$. The only wps relevant to x in the descendant database of $database(3)_{21}$ is B_{31}. So, t_{32} will see that wps. Then A_{32} must be equal to or a superset of B_{31}.

The Algorithms

The algorithms used to decide whether a certain lock can be given or not will now be presented. Note that the goal is to give the logical idea, and not any programming details.

Can a certain read lock be given? Assume that t_{ir}, executing in $database(i)_{(i-1)j}$, aquires a *read* lock on the object x. The system must do as shown in figure 5.13 in order to decide whether the lock can be given or not. The subalgorithm used by the algorithm shown in figure 5.13 is presented in figure 5.7.

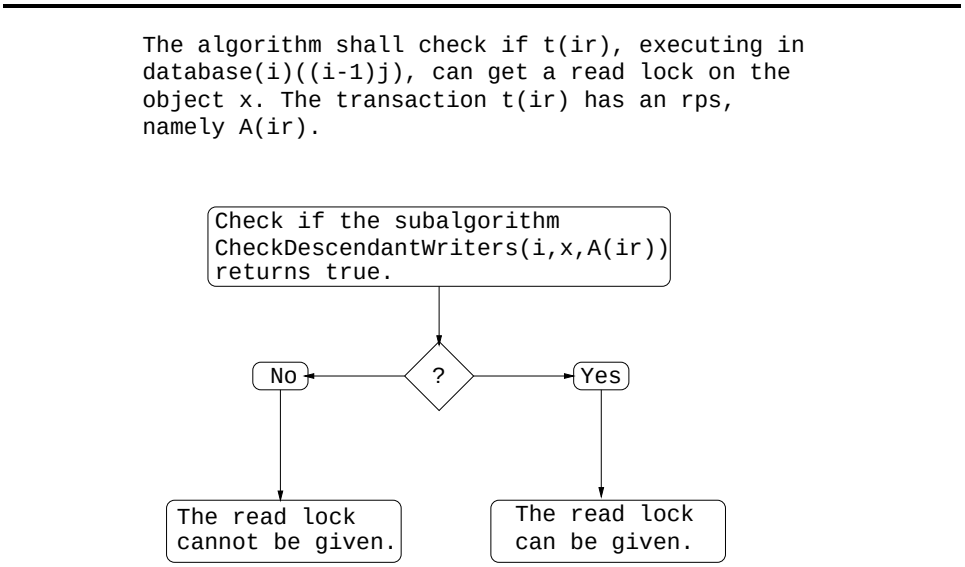

Figure 5.13: The algorithm used to decide if a transaction can get a read lock in solution 2.

The explanation that follows is very detailed, so the experienced reader may skip to page 66.

Consider figure 5.13. The algorithm must check if there are held any DB- or write lock(s) on x in the descendant database(s) of $database(i)_{(i-1)j}$ that are incompatible with t_{ir}'s desired read lock. The subalgorithm *CheckDescendantWriters*, shown in figure 5.7, checks this recursively as explained in solution 1.

If CheckDescendantWriters returns true, then it means that no DB- or write lock held on x in the descendant databases of $database(i)_{(i-1)j}$ is incompatible with t_{ir}'s desired read lock. Consequently, t_{ir}'s read lock can be given. However, if CheckDescendantWriters returns false, then it means that there exists at least one DB- or write lock which is incompatible with t_{ir}'s desired read lock. Then the read lock cannot be given. The details of the subalgorithm CheckDescendantWriters are explained in solution 1.

Can a certain write lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *write* lock on the object x. The system must do as shown in figure 5.8 in order to decide whether the lock can be given or not. This is the same algorithm as was used in solution 1, and it uses the subalgorithm presented in figure 5.9.

An explanation of why the algorithm used by solution 1 also is the right algorithm to be used by solution 2 follows.

When a transaction t_{iw}, executing in $database(i)_{(i-1)j}$, requests a write lock on an object x, then the first step for the system in order to decide whether the lock can be given or not is to check if there is another transaction holding a DB- or write lock on that object in $database(i)_{(i-1)j}$. If there is, then the lock cannot be given. If there is not, then we know that $database(i)_{(i-1)j}$ is the leaf database in the database tree relevant to x. Consequently, possibly existing readers of x exist in the *ancestor* databases of $database(i)_{(i-1)j}$.

Since a transaction with read access to an object shall see all possibly existing wpss in the *descendant* databases of the one it is executing in, a transaction with write access to a certain object must have a write lock that is compatible with all possibly held read locks in the *ancestor* databases of the one it is executing in. This means that the system in order to find out whether t_{iw}'s desired write lock on x can be given or not must check all possibly existing readers of x in the ancestor databases of $database(i)_{(i-1)j}$.

This is exactly what the algorithm used by solution 1 does.

Can a certain DB lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *DB* lock on the object x. If it is assumed that t_{iw} already has a write lock on the object x, then this lock can be given right away since the compatibility properties of write locks and DB locks are the same.

Discussion of Solution 2 and Introduction of Parameter Usage Rules

A discussion of solution 2 follows. In this solution we have seen that it is required that a read lock held on the object x by the transaction t_{ir}, executing in $database(i)_{(i-1)j}$, must be compatible with all possibly held

DB- and write locks on x in the *descendant* databases of $database(i)_{(i-1)j}$. Then the arbitrary transaction t_{ir}, holding a read lock on the object x in $database(i)_{(i-1)j}$, must have an rps which is the union of the wpss relevant to the object x in the descendant database(s) of $database(i)_{(i-1)j}$. If $database(i)_{(i-1)j}$ has superior databases, then the possibly held DB locks in those may be incompatible with t_{ir}'s read lock. This means that a transaction holding a read lock on a certain object will not get all information given in wpss belonging to transactions holding DB- or write locks on that object, but only the information given in the possibly existing wps relevant to the object in the database it is executing in and that given in the possibly existing wpss relevant to the object in the databases it is a foreigner to.

If no transactions have DB- or write locked x in these databases, then t_{ir} can access x in whatever parameterized read mode it wants, and no wps information will be given to it about x.

Also in this solution there may be quite a lot of checking to do in order to find out whether a certain lock can be given or not. It is necessary to check if any conflicting locks are held on the actual object in the descendant database(s) of the database where the lock requester executes. So if the actual database has many descendant databases containing the actual object, then the amount of checking will be large.

Even though there may be less checking in this solution than in solution 1, it may still become quite complicated.

If it *is* desirable to let readers see the wpss relevant to an object in the descendant databases of the one it executes, then restrictions on parameter set usage in subdatabases can be made in order to get a less complicated checking process when the system shall decide whether a certain lock can be given or not.

The rules presented in figure 5.14, restrict the parameter set usage in subdatabases. As we will se below, when these rules are enforced, it will be easier to decide whether a certain lock can be given or not.

The new Algorithms

The new algorithms used to decide whether a certain lock can be given or not will now be presented.

Can a certain read lock be given? Assume that t_{ir}, executing in $database(i)_{(i-1)j}$, aquires a *read* lock on the object x. The system must do as shown in figure 5.15 in order to decide whether the lock can be given or not.

A detailed explanation of the algorithm follows. Consider figure 5.15. The algorithm checks if there is a transaction t_{iw} in $database(i)_{(i-1)j}$ that has DB- or write locked x. If it is not, then the read lock can be given. Otherwise, it must be checked whether t_{ir}'s rps is equal to or a superset of t_{iw}'s wps. If it is not, then the read lock cannot be given. But if it is, then

```
Let the subdatabase transactions t(iw) and t(ir)
execute in database(i)((i-1)j).
Let t(iw) have a DB- or write lock on an arbitrary
object x, and let t(ir) hold a read lock on the
same object.


1. The wps of t(iw) must be equal to or a subset of
   database(i)((i-1)j)'s wps B((i-1)j).

2. The rps of t(ir) must be equal to or a superset
   of t(iw)'s wps B(ij).
```

Figure 5.14: The rules that restrict the subdatabase parameter usage in solution 2.

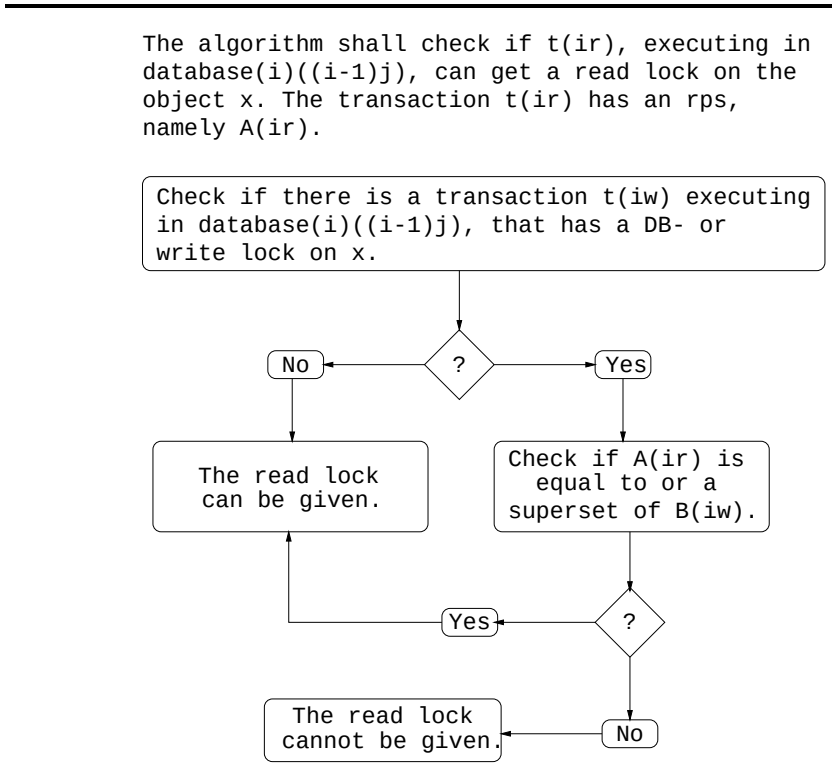

Figure 5.15: The algorithm used to decide if a transaction can get a read lock in solution 2, when the parameter usage rules for subdatabases shown in figure 5.14 are enforced.

the read lock can be given without any further checking. This is because the other possibly existing DB- or write locks on the actual object in the descendant databases of $database(i)_{(i-1)j}$ must have wpss that are equal to or supersets of t_{iw}'s wpss according to rule 1 shown in figure 5.14.

Can a certain write lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *write* lock on the object x. The system must do as shown in figure 5.16 in order to decide whether the lock can be given or not.

A detailed explanation of the algorithm follows. Consider figure 5.16. The algorithm checks first if there are another transaction in $database(i)_{(i-1)j}$ that holds a DB- or write lock on the object x. If there is, then the write lock cannot be given. If there is not, then possibly existing read locks held on x in $database(i)_{(i-1)j}$ must be checked in order to find out if they are incompatible with t_{iw}'s desired write lock. It is not necessary to check the foreign readers of x. This is because a foreigner has an rps that is equal to or a superset of the wps of the transaction holding a DB lock on x in the database where it executes (rule 2 shown in figure 5.14) and because t_{iw}'s wps is equal to or a subset of all wpss of ancestor databases (rule 1 shown in figure 5.14). So, if all possibly existing readers of x in $database(i)_{(i-1)j}$ hold read locks that are compatible with t_{iw}'s desired write lock, then it can be given. Otherwise, if at least one of these readers hold a read lock that is incompatible with t_{iw}'s desired write lock, then it cannot be given.

The Consequences of Enforcing the Parameter Usage Rules

The consequences of enforcing the rules presented in figure 5.14 will now be discussed. Recall that solution 2 is interesting if it is desirable that a transaction holding a read lock on a certain object, shall see the wpss relevant to that object in the descendant databases of the database it executes in. In order to get less complicated algorithms for deciding whether a certain lock should be given or not, the rules shown in figure 5.14 were presented.

By enforcing these parameter set usage rules, the algorithms in solution 2 become easier. Instead of checking all wpss relevant to the object x in the descendant databases of $database(i)_{(i-1)j}$ when the transaction t_{ir} aquires a read lock on it, it is sufficient to check only the wps of the possibly existing transaction, t_{iw}, holding a DB- or write lock on the actual object in $database(i)_{(i-1)j}$.

When t_{iw} aquires a write lock and it is found that no other transaction in $database(i)_{(i-1)j}$ holds a DB- or write lock on x, then instead of checking all possibly existing read locks held on the actual object in the ancestor databases of $database(i)_{(i-1)j}$, it is sufficient to check the possibly existing readers in $database(i)_{(i-1)j}$.

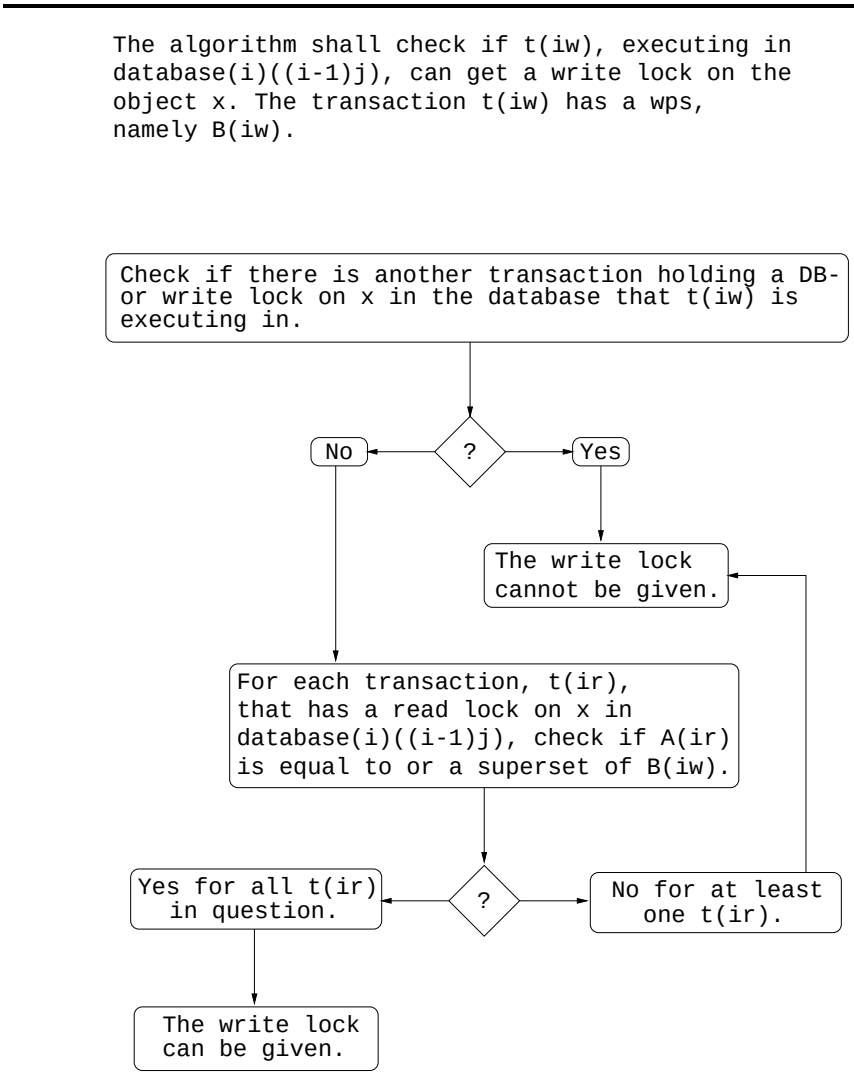

Figure 5.16: The algorithm used to decide if a transaction can get a write lock in solution 2, when the parameter usage rules for subdatabases shown in figure 5.14 are enforced.

A DB lock can as before be given right away, assumed that write locks are held initially to the DB lock request on the actual object(s).

As we also saw in solution 1, while the algorithms become easier less freedom is gained for subdatabase transactions in their parameter set usage. A subdatabase writer t_{iw}, executing in $database(i)_{(i-1)j}$, can only use a wps that is equal to or a subset of the wps of $database(i)_{(i-1)j}$, and a subdatabase reader t_{ir}, executing in $database(i)_{(i-1)j}$, must have an rps that is equal to or a superset of the wps of the possibly existing transaction holding a DB- or write lock on the actual object in $database(i)_{(i-1)j}$.

5.2.5 Solution 3

Assume that there is a transaction t_{ir}, executing in $database(i)_{(i-1)j}$, that holds a read lock on the object x. Solution 3 requires that only the possibly existing wps relevant to x in $database(i)_{(i-1)j}$ shall be visible for t_{ir}. Then if x is DB- or write locked in $database(i)_{(i-1)j}$, then that DB- or write lock have to be compatible with t_{ir}'s read lock.

Example

As an example, we will again consider figure 5.1. The transaction t_{12} is executing in $database(1)$. The wps relevant to x in $database(1)$ is B_{11}, and that is the only wps t_{12} will see. Then A_{12} must be equal to or a superset of B_{11}.

Let us consider t_{22} next. This transaction executes in $database(2)_{11}$. The wps relevant to x in $database(2)_{11}$ is B_{21}, and that is consequently the only wps t_{22} will see. So, A_{22} must be equal to or a superset of B_{21}.

Finally consider t_{32} executing in $database(3)_{21}$. The wps relevant to x in $database(3)_{21}$ is B_{31}. Then t_{32} will see this wps and A_{32} must be equal to or a superset of B_{31}. Note that if x was not DB- or write locked in $database(3)_{21}$, then t_{32} could get a read lock on x in whatever parameterized mode it would like.

The Algorithms

The algorithms used to decide whether a certain lock can be given or not will now be presented. Note again that the goal is to give the logical idea, and not any programming details.

Can a certain read lock be given? Assume that t_{ir}, executing in $database(i)_{(i-1)j}$, aquires a *read* lock on the object x. The system must do as shown in figure 5.17 in order to decide whether the lock can be given or not.

A detailed explanation of the algorithm follows. Consider figure 5.17. The algorithm checks if there is a transaction t_{iw} in $database(i)_{(i-1)j}$ that has a DB- or write lock on x. If there is not, then it means that no other

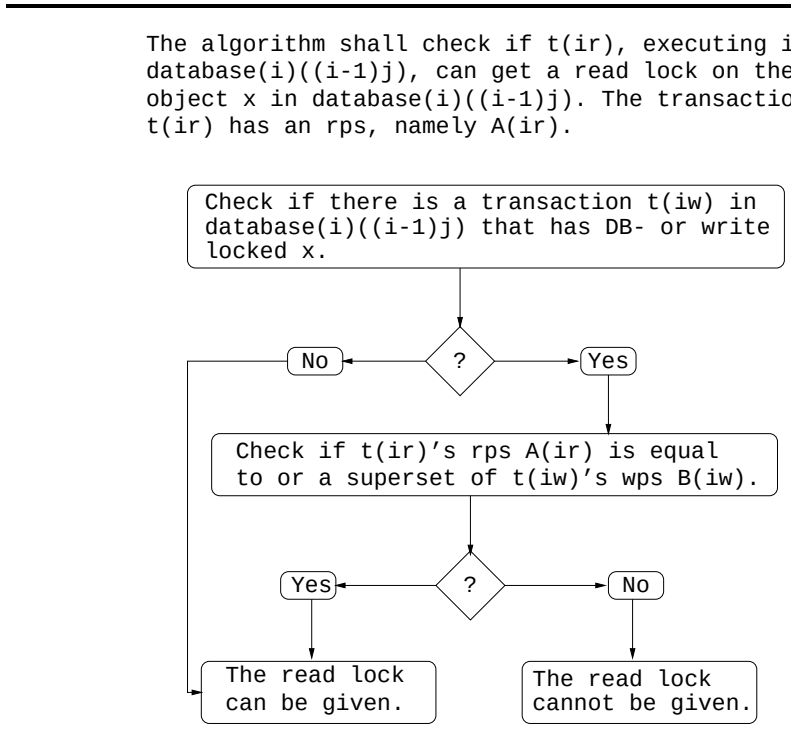

Figure 5.17: The algorithm used to decide if a transaction can get a read lock in solution 3.

transaction in $database(i)_{(i-1)j}$ has a lock that is incompatible with t_{ir}'s desired read lock, and it can therefore be given. But if there is another transaction t_{iw}, that holds a DB- or write lock on x in $database(i)_{(i-1)j}$, then the algorithm checks whether t_{ir}'s rps is equal to or a superset of t_{iw}'s wps or not. If it is, then it means that no other transaction in $database(i)_{(i-1)j}$ has a lock that is incompatible with t_{ir}'s desired read lock, and it can therefore be given. But if $A_{ir} \not\supseteq B_{iw}$, then there exists a transaction which has a DB- or write lock that is incompatible with t_{ir}'s desired read lock in $database(i)_{(i-1)j}$. Consequently, the read lock cannot be given.

Can a certain write lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *write* lock on the object x. The system must do as shown in figure 5.18 in order to decide whether the lock can be given or not.

A detailed explanation of the algorithm follows. Consider figure 5.18. The algorithm checks first if there is a transaction t_{iw} in $database(i)_{(i-1)j}$ that has a DB- or write lock on x. If there is, then the write lock cannot be given. However, if there is no such transaction, then the next step is to check if there exist one or more transactions in $database(i)_{(i-1)j}$ that hold a read lock incompatible with t_{iw}'s desired write lock. So, each transaction t_{ir}, executing in $database(i)_{(i-1)j}$, that holds a read lock on x must be checked in order to find out if $A_{ir} \supseteq B_{iw}$. If this is true for all the actual readers, then the write lock can be given. Otherwise, it cannot.

Can a certain DB lock be given? Assume that t_{iw}, executing in $database(i)_{(i-1)j}$, aquires a *DB* lock on the object x. If it is assumed that t_{iw} already has a write lock on the object x, then this lock can be given right away since the compatibility properties of write locks and DB locks are the same.

Discussion of Solution 3

A discussion of solution 3 follows. In this solution we have seen that it is only required that a read lock held on the object x in $database(i)_{(i-1)j}$ must be compatible with the possibly held DB- or write lock in $database(i)_{(i-1)j}$. If $database(i)_{(i-1)j}$ has superior and/or inferior databases, then the possibly held DB- and write locks in those databases may be incompatible with the actual read lock.

Then the transaction holding a read lock on the object x in $database(i)_{(i-1)j}$ must have an rps which is equal to or a superset of the possibly existing wps relevant to object x in $database(i)_{(i-1)j}$. This means that a transaction holding a read lock on a certain object will not get all information given in wpss belonging to transactions holding DB- or write locks

The algorithm shall check if t(iw), executing in database(i)((i-1)j), can get a write lock on the object x. The transaction t(iw) has a wps, namely B(iw).

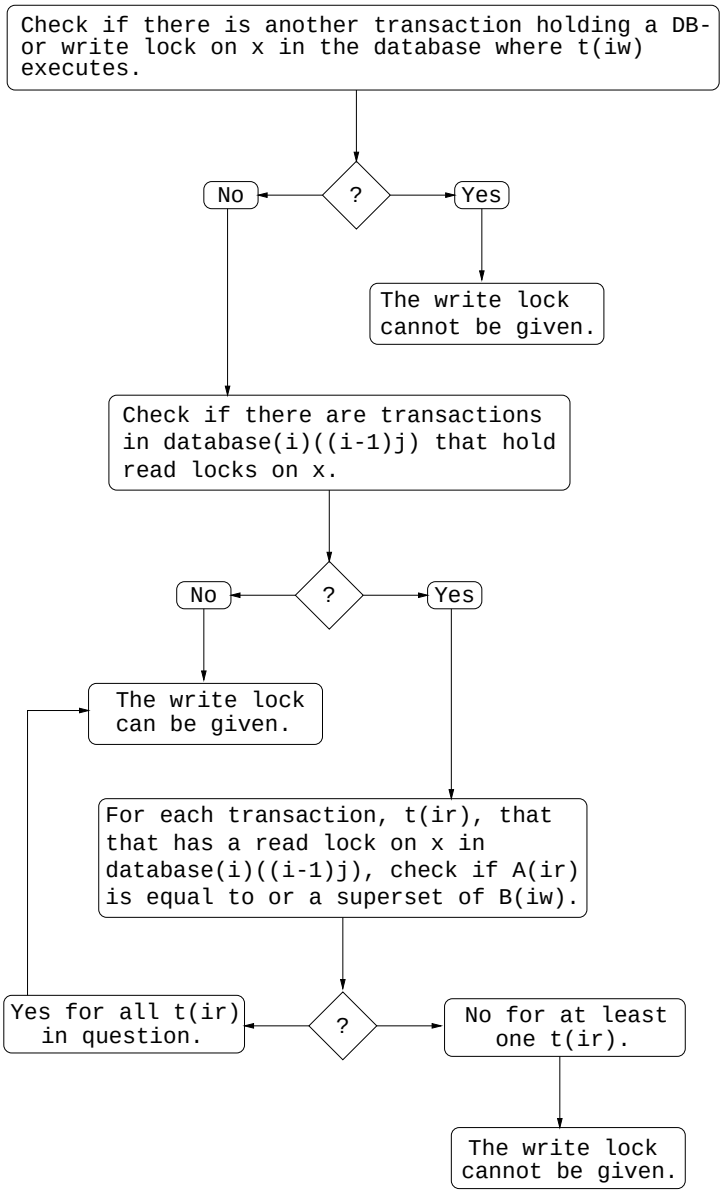

Figure 5.18: The algorithm used to decide if a transaction can get a write lock in solution 3.

on that object, but only the information given in the possibly existing wps relevant to the object in the database it is executing in.

In this solution, it is only necessary to consider the database where the actual lock requester executes in order to decide whether a certain lock can be given or not.

If no transaction has DB- or write locked x in $database(i)_{(i-1)j}$, then t_{ir} can access x in whatever parameterized read mode it wants, and no wps information will be given to it about x.

In this solution less information is given to readers since they only get to see possibly *one* wps relevant to a certain object. But subdatabase transactions get more freedom in their parameter usage, and the algorithms used to decide whether a certain lock can be given or not are simple.

5.3 Summary

In this chapter parameterized DB locks were considered. It has been assumed that each transaction at most has one wps and one rps, that transactions executing in subdatabases and those holding DB locks in the top-level database *must* have parameter sets, and that a subdatabase has the same wps as its creator.

The problem of which wpss readers should see when there are more than one wps relevant to an object was discussed. Three solutions were presented and discussed.

Solution 1 required that an arbitrary reader should see *all* wpss relevant to an object, causing readers to get all information about their objects given in wpss. This resulted in quite complicated algorithms where many locks in different databases had to be checked before the system was able to decide whether a certain lock could be given or not. In order to reduce the amount of checking, two rules for parameter set usage in subdatabases were introduced. When these rules were enforced the algoritms became easier, but at the same time subdatabase transactions got less freedom in their parameter set usage. In this solution readers got much information about their objects.

Solution 2 required that an arbitrary reader should see the possibly existing wpss relevant to an object in the *descendant* databases of the one where the reader was executing, causing readers to get information about their objects given in wpss existing in the databases where they were executing, and in those to they were foreigners. With this solution the possibly existing information about a reader's objects given in wpss belonging to superior databases of the database where the reader was executing would not be given to it. This solution resulted in quite complicated algorithms too, and again two rules for parameter set usage in subdatbases were introduced. The algorithms became easier while the subdatabase transactions got less freedom in their parameter set usage. In this solution the readers got less

information about their objects than in solution 1 unless they executed in the top-level database.

Solution 3 required that an arbitrary reader should see the possibly existing wps relevant to an object in the database where the reader was executing, causing readers to only get information about their objects given in wpss existing in the databases where they were executing. With this solution the possibly existing information given in wpss belonging to superior and inferior databases of the reader's database would not be given to the reader. The algorithms in this solution were simple, and subdatabase transactions got much freedom in their parameter set usage. In this solution readers got little information about their objects compared to solution 1 and 2 unless only one wps was relevant to each object in the hole database tree.

Chapter 6

Dynamic Modification of Concurrency Levels

We will in this chapter consider issues concerning dynamic modification of a transaction's concurrency level.

6.1 What are Concurrency Levels?

A transacation's *concurrency level* refers to both its access mode parameters and its isolation level (Anfindsen, 1997, page 43).

A transaction's *isolation level* gives a guaranty for how long the transaction's read locks are held. In other words, it specifies when other transactions can overwrite data items read by the transaction in question. In this thesis it is assumed that rigorous 2PPL is enforced, causing an arbitrary read lock to be held until a transaction has aborted, rolled back to a point before the lock was taken, or committed. So only *one* isolation level is considered in this context, and consequently the concept of *dynamic modification of concurrency levels* will mean *dynamic modification of parameter sets.*

For more information about isolation levels see (Anfindsen, 1997, pages 14–16).

6.2 Presentation of the Problem

(Anfindsen, 1997, page 108) writes the following:

> *If a data item locked in w(B) mode is read by another transaction in r(A) mode, what should then happen if the first transaction attempts to change lock mode from w(B) to w(C)? Should this be prevented? If not, should the reader be notified? Or perhaps the outcome should depend on whether or not* $C \subseteq A$*? And should*

there be rules limiting how lock parameters can change in general?

These questions will be considered in more detail in this section, and ways of changing lock modes instead of preventing them will be sought.

(Anfindsen, 1997, page 55) also writes the following:

... It's desirable to allow a transaction to dynamically modify its own concurrency level. For example, when a designer reaches a new level of approval for his or her design objects, all write parameters for those objects could be modified....

If a transaction wants to change the notion of conflict and communicate a new status or quality of its data items to other transactions, then it must change its write parameter set from one set to another. If a reading transaction wants to change the notion of conflict, then it must change its read parameter set from one set to another.

6.2.1 Example

As a concrete example consider once again Bob and Alice. Bob is working on the hydraulic system and Alice needs to see his uncommitted work in order to do her work on the landing gear. The hydraulic system data is contained in the object H.

Bob's transaction begins and then Alice's transaction begins. Bob's transaction write locks H in a w(ID) mode, does some work, and writes it back to the database. Then Alice read locks H in an r(ID) mode and reads the object. After a while Bob wants to change the write mode to w(CD), but if he does, then his new write mode will be incompatible with Alice's read mode r(ID) held on the object H.

6.2.2 Can a Parameter Change be Executed?

We will now consider what the system must do in order to decide whether a parameter set change can be executed or not when it is required that a transaction's rps always must be equal to or a superset of the wpss *that are supposed to be visible* for the transaction. What wpss that are supposed to be visible for the transaction depends on the choice of solution discussed in section 5.2.

Assume that t_{ir} is executing in $database(i)_{(i-1)j}$ and that it holds read locks on some objects. Then assume further that t_{ir} requests an rps change from A_{ir1} to A_{ir2}. (The third subscript is to indicate what order the rps has among the actual transaction's rpss.)

Then the system must, for *each* object read locked by t_{ir}, consider the possibly held DB- and write locks on that object which is required to be compatible with t_{ir}'s read mode.

For an example, assume that t_{22} executing in $database(2)_{11}$, shown in figure 6.1, holds read locks on the objects w, x, y, and z, and that it requests an rps change from A_{221} to A_{222}.

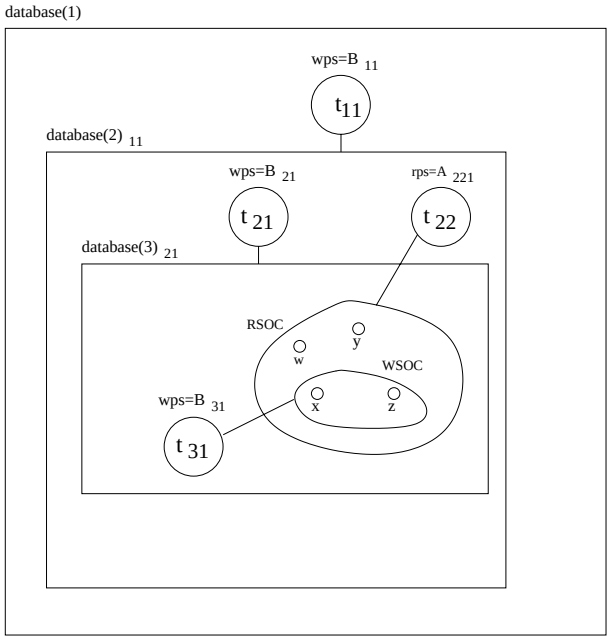

Figure 6.1: The transaction t_{22} holds read locks on the objects w, x, y, and z.

Assume that solution 1 presented in section 5.2.3 is followed. Then the system must first consider e.g. the object w read locked by t_{22}. This object is DB locked by t_{21} and t_{11}, and consequently the wpss of these two transactions must be checked. Next e.g. the object x, also read locked by t_{22}, must be considered. This object is write locked by t_{31} and DB locked by t_{21} and t_{11}, so the wpss of these three transactions must be checked. Further the two last objects y and z must be considered in a similar way.

If, for each object, the relevant possibly existing DB- and write locks are compatible with t_{22}'s new read mode, then it is ok to execute the rps change. But if at least one relevant DB- or write lock is incompatible with t_{22}'s desired new read mode, then the rps change cannot be executed.

Assume that t_{iw} is executing in $database(i)_{(i-1)j}$ and that it holds a DB lock, or several write locks on some objects. Then assume further that t_{iw} requests a wps change from B_{iw1} to B_{iw2}.

Then the system must, for *each* object that is covered by t_{iw}'s DB lock or one of t_{ir}'s write locks, consider the possibly held read locks on that object which is required to be compatible with t_{iw}'s DB- or write mode. If, for

each object, the relevant possibly existing read locks are compatible with t_{iw}'s new DB- or write mode, then it is ok to execute the wps change. But if at least one read lock is incompatible with t_{iw}'s desired new DB- or write mode, then the wps change cannot be executed.

6.2.3 Consequences

When the system shall decide whether a parameter set change can be executed or not, it may result in a lot of checking if many objects, subdatabases and transactions are involved.

There may be a lot more checking to do when a *parameter set change* is requested than when a *new lock* is requested.

This is because when e.g. a transaction t_{iw}, holding write locks on some objects in $database(i)_{(i-1)j}$, wants to change its wps, then not only the locks held on *one* object must be considered but the locks held on *all* objects write locked by t_{iw}.

For example, assume that solution 1, presented in section 5.2.3, is chosen to specify which wpss readers shall see. Then if t_{iw} has write locked n objects and wants to do a wps change, the subrutine CheckAncestorReaders presented in section 5.2.3 would have had to be called n times. But if t_{iw} requested a new write lock, then the subrutine in question would have had to be called only once. So, if n is large, if many transactions have read locks on the objects write locked by t_{iw}, and if these objects are contained in many databases, then the amount of checking may be large.

Therefore, it is interesting to consider how this checking can be reduced.

6.2.4 A wps Change Requirement

We will next consider a wps change requirement that reduces the necessary checking that must be done when the system shall decide whether a wps change can be executed or not.

It will be required, as above, that a transaction's rps always must be equal to or a superset of all wpss that are supposed to be visible for the transaction, and *in addition* it is required that a wps only can be changed to a *subset* of itself.

Then if e.g. a transaction t_{iw}, holding write locks on some objects in $database(i)_{(i-1)j}$, wants to change its wps, no checking of other locks held on the actual objects is necessary in order to decide whether the requested wps change can be executed or not. This is because if the new wps is a subset of the current wps, then all rpss relevant to the actual objects are supersets of the new wps since they already are equal to or supersets of the current wps. However, if t_{iw} requests a wps change to a wps which is not a subset of the current wps, then the wps change can not be executed.

This extra requirement has a consequence on parameter set usage in subdatabases as we now will consider. When a transaction can only change its wps to a subset of its current wps, then it must make sure that it starts out with a wps that contains *all* parameters which is to become valid in the future. Since a wps must contain all parameters which is to become valid in the future, there must be a way to specify which parameter(s) that are currently valid. E.g. if it can be defined an ordering between the parameters, then either the system or the transaction owners must know about this in order to make it possible for the transaction owners to interpret the information given in wpss correctly. E.g. the parameter ID could be stronger than the parameter CD. Then if a wps contains these two parameters, the strongest parameter, namely ID, is valid while CD is a state that the actual objects will reach in the future. When the wps is changed from {ID,CD} to {CD}, then CD is obviously valid.

So, with the requirement of only allowing a transaction to change to a subset of its current wps, no checking of read locks is necessary in order to decide whether a wps change can be executed or not. But parameter sets must be used and interpreted in a certain way as pointed out above.

6.3 Summary

In this chapter dynamic modification of parameter sets has been discussed. It was required that a transaction's rps always had to be equal to or a superset of the union of the wpss of transactions that held DB- or write locks on at least one of the objects on which the reader in question held a read lock.

It could be quite complicated to decide whether a parameter set change could be executed or not because of all the locks that had to be checked. Therefore, we found that it was interesting to reduce this checking when a parameter set change is requested.

One proposal as to how the checking could possibly be reduced in order to decide whether a wps change could be executed or not was presented and discussed. The proposal was to restrict a transacation to only be allowed to change its wps to a subset of its current wps. Then a transaction had to start to use a wps that contained all parameters that was to be used in the future, and there had to be a way to specify which parameter(s) that were currently valid in order to make it possible for transaction owners to interpret wpss correctly.

List of Figures

Bibliography

Anfindsen, O. J. (1997). *Apotram—An Application-Oriented Transaction Model*. PhD thesis, University of Oslo, Norway. Pages 8–9 and 29–30.

Bernstein, Hadzilacos, and Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company.

Davies, C. T. (1973). Recovery semantics for a db/dc system. *Proceedings of the ACM National Conference 28*.

Elmagarmid (1992). *Database Transaction Models for Advanced Applications*. The Morgan Kaufmann Series In Data Management Systems. Morgan Kaufmann Publishers, Inc.

Elmagarmid, A. K., Leu, Y., and Mullen, J. G. (1992). Introduction to advanced transaction models. In Elmagarmid, K., editor, *Database Transaction Models for Advanced Applications*, chapter 2, pages 34–52. Morgan Kaufmann Publishers, Inc.

Elmasri, R. and Navathe, S. B. (1994). *Fundamentals on Database Systems*. The Benjamin/Cummings Publishing Company, Inc., second edition.

Eswaran, K. P., N, G. J., A, L. R., and L, T. I. (1976). The notions of consistency and predicate locks in a database system. In *Communications of the ACM*, pages 624–633.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc.

Kaiser, G. E. (1995). Cooperative transactions for multiuser environments. In Kim, W., editor, *Modern Database Systems: the object model, interoperability, and beyond*, chapter 20, pages 409–433. Addison-Wesley Publishing Company.

Kaiser, G. E. and Pu, C. (1992). Dynamic restructuring of transactions. In Elmagarmid, K., editor, *Database Transaction Models for Advanced Applications*, chapter 8, pages 266–295. Morgan Kaufmann Publishers, Inc.

Moss, J. E. B. (1981). *Nested transactions—an approach to reliable distributed computing*. PhD thesis, The MIT Press, Cambridge, Mass.

Nodine, M. H., Ramaswamy, S., and Zdonik, S. B. (1992). A cooperative transaction model for design databases. In Elmagarmid, K., editor, *Database Transaction Models for Advanced Applications*, chapter 3, pages 54–85. Morgan Kaufmann Publishers, Inc.

Normann, R. and Ressem, J. E. (1998). Relasjonsdatabaser, databehandling i in114. Page 6.

Reed, D. P. (1978). *Naming and synchronization in a decentralized computer system*. PhD thesis, MIT Dept of Elect Eng and Comp Sci.

Silberschatz, Korth, and Sudarshan (1997). *Database System Concepts*. McGraw-Hill computer science series. McGraw-Hill, third edition.

Unland, R. and Schlageter, G. (1992). A transaction manager development facility for non standard database systems. In Elmagarmid, K., editor, *Database Transaction Models for Advanced Applications*, chapter 11, pages 400–466. Morgan Kaufmann Publishers, Inc.

Weikum, G. and Schek, H.-J. (1992). Concepts and applications of multilevel transactions and open nested transactions. In Elmagarmid, K., editor, *Database Transaction Models for Advanced Applications*, chapter 13, pages 516–553. Morgan Kaufmann Publishers, Inc.